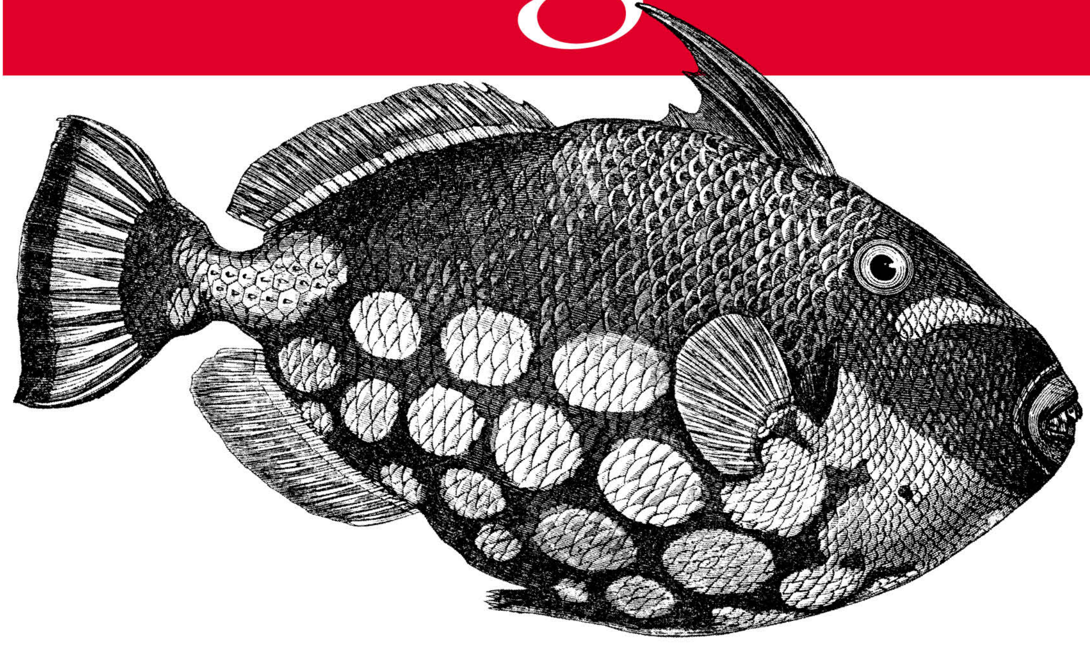
图灵程序设计丛书

Scaling MongoDB
50 Tips and Tricks for MongoDB Developers

深入学习 MongoDB



[美] *Kristina Chodorow* 著
巨成 程显峰 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

图灵程序设计丛书

深入学习MongoDB

Scaling MongoDB
50 Tips and Tricks for MongoDB Developers

[美] Kristina Chodorow 著
巨成　程显峰 译

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北　京

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会聚集了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创 新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

图书在版编目（C I P）数据

深入学习MongoDB / (美) 霍多罗夫 (Chodorow,K.)
著 ; 巨成, 程显峰译. -- 北京 : 人民邮电出版社,
2012.1
（图灵程序设计丛书）
ISBN 978-7-115-27211-9

Ⅰ. ①深… Ⅱ. ①霍… ②巨… ③程… Ⅲ. ①关系数
据库系统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2011)第267232号

内 容 提 要

本书分两部分，分别对应 O'Reilly 公司出版的 *Scaling MongoDB* 和 *50 Tips and Tricks for MongoDB Developers* 两本书的内容。第一部分全面讲解了有关建立和使用集群的内容，不仅从应用开发人员的角度讲解了 MongoDB 的使用，而且从运维方面介绍了集群的管理。其中内容包括通过分片设置 MongoDB 集群，分片的工作原理，查询和更新数据，操作、监控和备份集群，错误处理。第二部分依次从应用设计、实现、优化、数据安全和管理方面介绍了使用 MongoDB 构建应用的技巧，内容包括范式化与反范式化的利弊权衡，复制组的故障恢复等。

本书适合所有 MongoDB 用户阅读参考。

图灵程序设计丛书
深入学习MongoDB

◆ 著　　[美] Kristina Chodorow
译　　巨　成　程显峰
责任编辑　卢秀丽
执行编辑　毛倩倩

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　印刷

◆ 开本：800×1000　1/16
印张：8.5
字数：150千字　　　　2012年 1月第 1版
印数：1－4 000册　　　2012年 1月北京第 1 次印刷
著作权合同登记号　图字：01-2011-8108号
ISBN 978-7-115-27211-9

定价：32.00元
读者服务热线：(010)51095186转604　印装质量热线：(010)67129223
反盗版热线：(010)67171154

版权声明

©2011 by O’Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O’Reilly Media, Inc. and Posts & Telecom Press, 2012. Authorized translation of the English edition, 2012 O’Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O’Reilly Media, Inc. 出版 2011。

简体中文版由人民邮电出版社出版，2012。英文原版的翻译得到 O’Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O’Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会聚集了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创 新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

目录

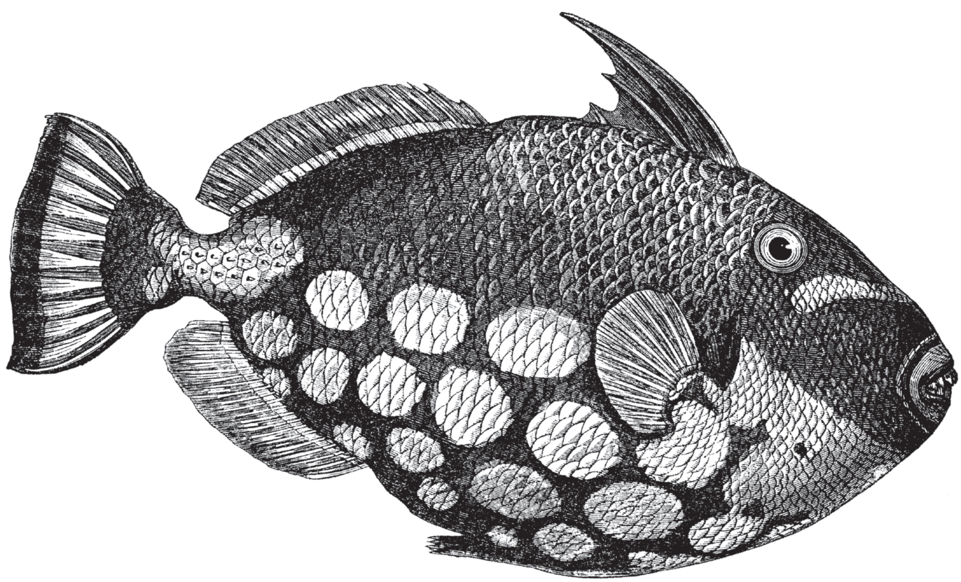
MongoDB 扩展技术

MongoDB 开发技巧 50 例

MongoDB扩展技术

Scaling MongoDB

［美］Kristina Chodorow 著
巨 成 译



前言

本书是为那些对分片感兴趣的 MongoDB 用户准备的，它全面讲述了如何建立和使用集群等内容。

本书不是对 MongoDB 的入门介绍。读者需要理解诸如文档、集合、数据库这些概念，知道如何读写数据，什么是索引，如何以及为什么要建立副本集。

如果你并不熟悉 MongoDB，大可不必担心，因为它很容易上手。市面上有一些有关 MongoDB 的书，包括本书作者参与编写的《MongoDB 权威指南》。你也可以查阅在线文档。

格式约定

本书使用了如下排版约定。

• 楷体

用于标记新名词。

• 等宽字体

用于程序代码，在段落中用于表示程序的组成部分，如变量或函数名、数据库、数据类型、环境变量、语句、关键字。

• 等宽粗体

命令或是其他应该由用户输入的内容。

• 等宽斜体

应该由用户提供的或根据上下文确定的值。

这个图标表示提示、建议或一般性的注解。

这个图标表示一个警告或警示。

使用示例代码

本书用于帮助你完成工作。通常，你可以在程序或文档中使用本书提供的代码。除非你在重新发布我们的大量代码，否则不需要联系我们来获得许可。比如，在程序中使用本书代码的一些片段是无需我们许可的。但是出售或再分发 O'Reilly 的图书示例光盘显然是需要授权的。引用本书或引用示例代码来回答问题是不需要授权的，但将本书的大量示例代码纳入产品的文档是需要授权的。

我们乐于见到你在使用时声明引用信息，但不强制要求。引用信息通常包括书名、作者、出版社和 ISBN，例如"*Scaling MongoDB* by Kristina Chodorow (O'Reilly). Copyright 2011 Kristina Chodorow, 978-1-449-30321-1"。

如果你认为对示例代码的使用需要授权，请通过这个邮箱联系我们：permissions@oreilly.com。

Safari®在线图书

Safari 在线图书是应需而变的数字图书馆。它能够让你非常轻松地搜索 7500 多种技术性和创新性参考书以及视频，以便快速地找到需要的答案。

订阅后你就可以访问在线图书馆内的所有页面和视频，在手机或其他移动设备上阅读，在新书上市之前抢先阅读，还能够看到尚在创作中的书稿并向作者反馈意见。复制粘贴代码示例、放入收藏夹、下载部分章节、标记关键点、做笔记甚至打印页面等有用的功能可以帮你节省大量时间。

这本书也在其中。欲访问本书的英文版电子版，或者由 O'Reilly 或其他出版社出版的相关图书，请到 http://my.safaribooksonline.com 免费注册。

我们的联系方式

请把对本书的评论和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreilly.com/catalog/9781449303211

中文书：

http://www.oreilly.com.cn/index.php?func=book&isbn=9787115272119

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

关于本书的更多信息、会议、资源中心和网络，请访问以下网站：

http://www.oreilly.com
http://www.oreilly.com.cn

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

第 1 章

欢迎来到分布式计算的世界

在《终结者》系列影片中，一个称作“天网”的人工智能生命向人类发动战争，年复一年地制造机器人和杀戮人类。这是大部分运维人员的梦想，当然不是指毁灭人类，而是指构建一个可以长时间运行而无需人工干预的分布式系统。遗憾的是，时至今日“天网”依旧是个幻想，因为设计好并维护其稳定持续运行，对一个分布式系统来说仍然是一件非常困难的事情。

单台数据库服务器的状态通常很简单：非启即停。但是如果再添一台服务器并把数据分开来，则这两台服务器之间会产生某种依赖。假设其中一台停机，对另一台会造成什么影响？你的应用程序能应付其中一台（或两台一起）停机的情况吗？如果两台都在运行但无法通信呢？又或是可以通信，但是速度非常非常慢呢？

随着更多节点被添加到集群里，这类问题会变得越来越多和复杂。如果集群中的一整部分无法与其他部分通信会发生什么？如果一部分机器崩溃了又会如何？如果整个数据中心都出问题了呢？突然之间，即使是创建一个备份也将变得异常困难。怎样为分布在集群中几十台机器上的 TB 级数据建立一致性快照，但又不会冻结正在使用这些数据的应用程序？

如果一台服务器可以满足需求，那就能避免很多问题。但是如果想要存储大量数据或者想以高于单服务器处理能力的频率来访问这些数据，则建立一个集群是不可避免的。MongoDB 的优势之一正是试图帮助你解决上面列出的许多问题。不过这并不像设置单个 *mongod* 实例（这又是什么？）那么简单。本书将向你展示如何一步步建立起一个健壮的集群，以及在这个过程中将遇到的各种挑战。

什么是分片

分片（sharding）是 MongoDB 用来将大型集合分割到不同服务器（或者说一个*集群*）上所采用的方法。尽管分片起源于关系型数据库分区，但它（像 MongoDB 的大部分方面一样）完全是另一回事。

和你可能使用过的任何分区方案相比，MongoDB 的最大区别在于它几乎能自动完成所有事情。只要告诉 MongoDB 要分配数据，它就能自动维护数据在不同服务器之间的均衡。当然，你得告诉 MongoDB 把服务器添加到集群中，不过只要这么做了，MongoDB 同样会确保新加入的服务器分得均等的数据。

分片主要是为了实现 3 个简单的目标。

让集群“不可见”

应用程序只要知道跟它打交道的是一个普通的 *mongod* 实例就够了。

为了实现这一目标，MongoDB 自带了一个叫做 *mongos* 的专有路由进程。*mongos* 坐镇集群大前方，对连上它的任何应用而言就像是一个普通的 *mongod* 服务器。*mongos* 会把请求正确无误地转发到集群中的一个或一组服务器上，接着再把获得的响应拼装起来发回给客户端。这样一来，客户端无需知道与其通信的是一台服务器还是一个集群。

不过由于集群本身的特性使然，也存在一些违背该抽象的特殊情况，这些特殊情况会在第 4 章中提到。

保证集群总是可读写

任何集群都无法保证永远可用（比如出现大范围停电之类的情况），但是在合理的条件下，永远都不应该出现用户无法读写数据的情况。在功能发生明显降级前，集群应当允许尽可能多的节点失效。

MongoDB 通过多种途径来确保最长的正常运行时间。集群的每一部分可以并且应当在其他服务器上（最理想的情况是在其他数据中心）有冗余的进程运行，以便当一个进程 / 机器 / 数据中心坏掉了，其他副本可以立即（自动地）接替坏掉的部分继续工作。

把数据从一台服务器迁移到另一台的过程中也存在着一个非常有趣的难题：在传输过程中如何保证对数据访问的持续性和一致性？我们已经找出了一些非常好的解决方案，不过有些超出本书的讲述范围。总之，MongoDB 采用了一些非常漂亮的技巧。

使集群易于扩展

当系统需要更多的空间或资源时，应当可以添加。MongoDB 支持按需扩充系统容量。有关增加（和移除）容量的更多内容参见第 3 章。

要实现这些目标，一个集群应该易于使用（就像使用单个节点一样）和易于管理（否则添加新的分片就不那么容易了）。MongoDB 能够轻而易举地让应用程序自然茁壮地成长。

第 2 章

理解分片

为了建立、管理或调试集群，需要了解分片的基本工作模式。本章包含这部分基本内容以便你理解所发生的事情。

2.1 分割数据

分片（shard）是集群中负责数据某一子集的一台或多台服务器。举个例子，如果有一个集群存储了 1 000 000 份代表网站用户的文档，则一个分片可能包含其中 200 000 位用户的信息。

一个分片可由多台服务器组成。如果分片包含不止一台服务器，则每台服务器都有一份完全相同的数据子集的副本（见图 2-1）。在生产环境中，一个分片通常是一个副本集（replica set）。

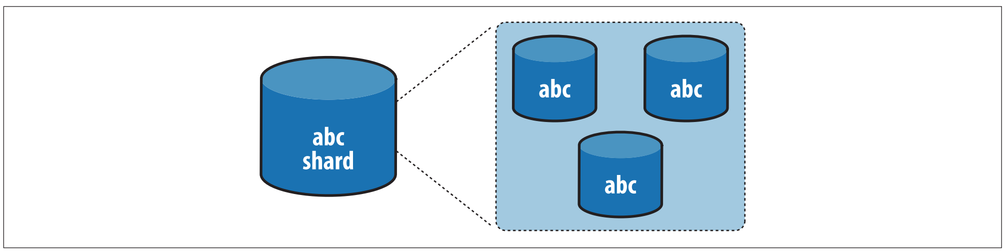


图 2-1：一个分片包含数据的某一子集。若某一分片包含多台服务器，则每台服务器拥有一份完整的数据副本

为了在分片间均匀地分配数据，MongoDB 会在不同分片间移动数据子集。它会根据片键（key）来决定移动哪些数据。例如我们可能选择按用户名（username）字段来划分用户集合。MongoDB 使用基于区间的方法进行划分，即按照给定区间将数据分割成不同块，例如 ["*a*", "*f*")。本章会使用标准的区间符号来描述区间。“[”和“]”表示闭区间，而“(”和“)”表示开区间。因此，有 4 种可能的区间：

x 在（a, b）中，当且仅当存在 x 使得 $a < x < b$。
x 在（a, b] 中，当且仅当存在 x 使得 $a < x \leqslant b$。
x 在 [a, b）中，当且仅当存在 x 使得 $a \leqslant x < b$。
x 在 [a, b] 中，当且仅当存在 x 使得 $a \leqslant x \leqslant b$。

MongoDB 中分片多用 [a, b）来表示区间范围，所以这会是最常见的形式。该区间可以表述为“从 a 开始且包含 a，到 b 为止但不包含 b”。

举个例子，比如说我们有一个用户名区间 `["a","f")`，那么 `"a"`、`"charlie"`，以及 `"ez-bake"` 都可能属于这一区间对应的集合，因为按字符串比较有 `"a"` ≤ `"a"`<

"charlie"<"ez-bake"<"f"。

该区间的范围到 "f" 为止但不包含 "f"，因此 "ez-bake" 可能属于这一集合，而 "f" 不属于。

2.1.1 分配数据

MongoDB 划分数据的方法有些不直观。为了理解这么做的理由，我们先使用初级方法，遇到问题时再寻找更好的方法。

1. 一分片一区间

分配数据到分片最简单的方法就是让每个分片负责一个区间的数据。所以，如果我们有 4 个分片，则很可能会得到如图 2-2 所示的设置。在这个例子里，我们假设所有用户名都以 "a" 到 "z" 之间的字母开头，其范围可以表示为区间 ["a","{")，{ 是 ASCII 码表中字母 z 后面的字符。

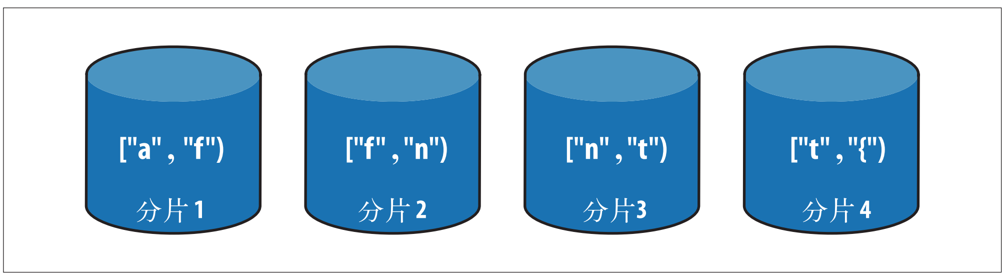


图 2-2：4 个分片，其区间分别是 ["a","f")、["f","n")、["n","t") 和 ["t","{")

这种分片体系非常简明易懂，但是在一个大型或繁忙的系统中却会带来许多不便。推想如此分片导致的后果，就很容易明白其中的道理。

假设许多用户用首字母在范围 ["a","f") 中的名字来注册。这会导致分片 1 较大，因此我们需要拿出它的一部分文档并将其挪到分片 2 上去。我们可以调整区间使分片 1（比方说）变成 ["a", "c")，使分片 2 变成 ["c", "n")（见图 2-3）。

目前好像没什么问题，但如果分片 2 因此过载该怎么办呢？假设分片 1 和分片 2 各有 500 GB 数据，而分片 3 和分片 4 各自只有 300 GB。那么按照这个分片方案，则最终要进行一连串复制：我们不得不把 100 GB 数据从分片 1 挪到分片 2，接着把 200 GB 数据从分片 2 挪到分片 3，最后再把 100 GB 数据从分片 3 挪到分片 4，算下来总共要挪动 400 GB 数据（见图 2-4）。考虑到所有数据移动都必须在集群中级联下去，可知额外移动的数据量很大。

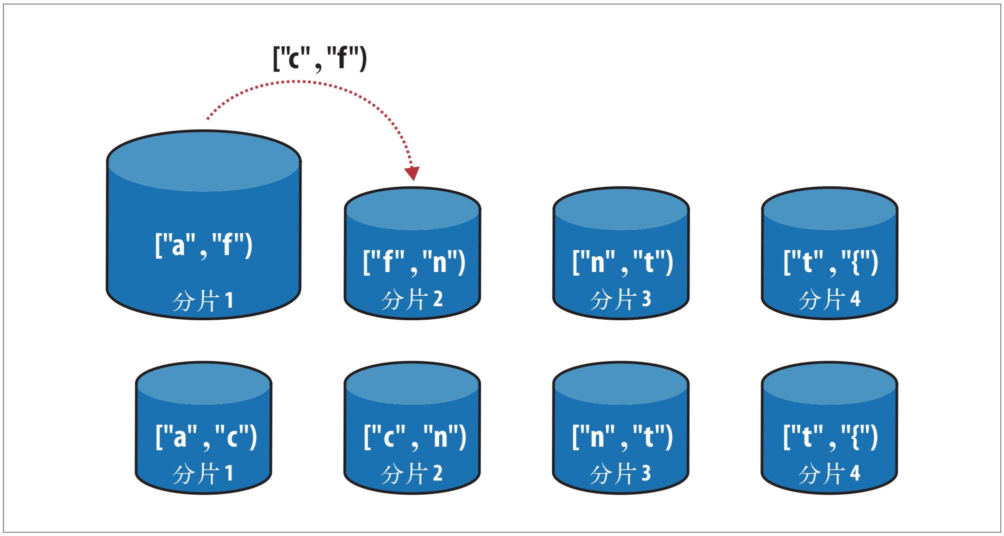


图 2-3：从分片 1 迁移部分数据到分片 2。分片 1 的区间缩小而分片 2 的区间扩大

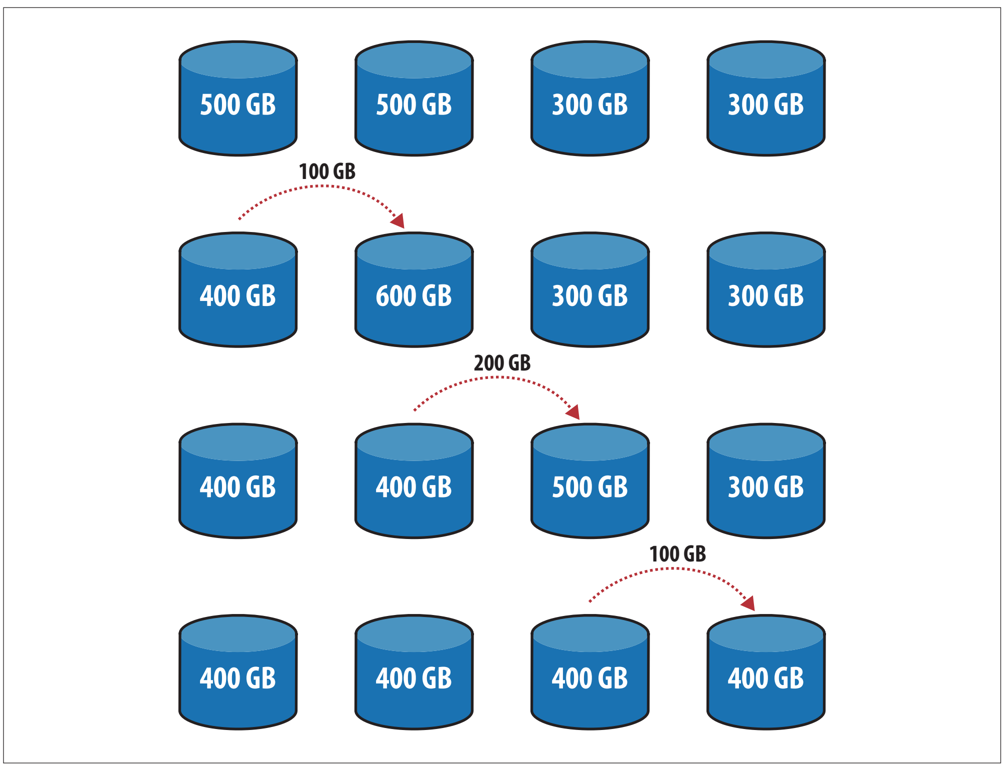


图 2-4：一分片一区间会导致级联效应：必须移动数据到“下一台”服务器上，即使不能改善平衡性

如果添加一个新分片又会怎样呢？假设这个集群继续工作下去，最终每个分片上都有了 500 GB 数据，然后我们添加一个新分片进来。现在我们不得不把 400 GB 数据从分片 4 挪到分片 5，将 300 GB 从分片 3 挪到分片 4，将 200 GB 从分片 2 挪到分片 3，将 100 GB 从分片 1 挪到分片 2（见图 2-5）。整整移动了 1 TB 数据！

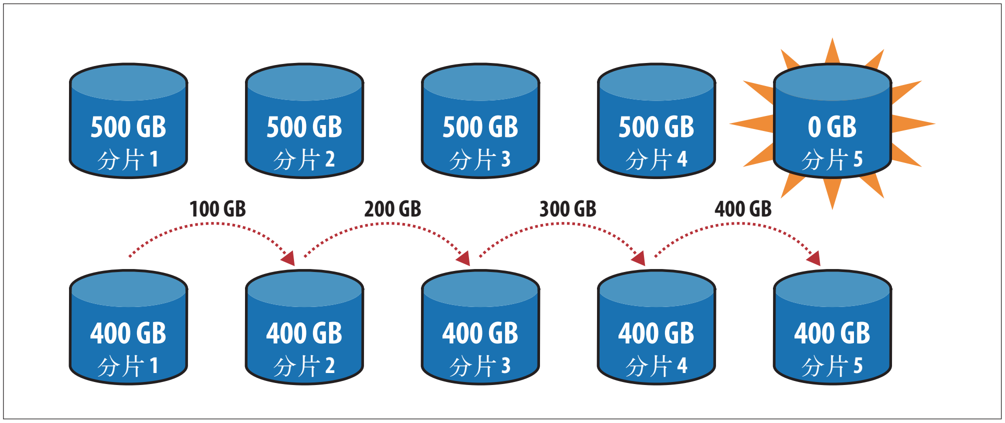


图 2-5：添加一台新服务器并使集群负载均衡。我们可以通过在集群的“中间”（分片 2 和分片 3 之间）添加服务器来降低所需迁移的数据量，但即使这样仍然需要迁移 600 GB 数据

随着分片数量和数据量的增长，这种级联效应只会持续恶化下去。因此 MongoDB 并不采用这种方法来分配数据，而是使每个分片包含多个区间。

2. 一分片多区间

让我们回头重新考虑如图 2-4 所示的情况，其中分片 1 和分片 2 各有 500 GB 数据，而分片 3 和分片 4 各有 300 GB 数据。这次我们允许每个分片包含多块区间。

我们可以把分片 1 上的数据划分为两个区间，使其中一个包含 400 GB 数据（比如说 `["a","d")`），另一个包含 100 GB 数据（`["d","f")`）。接着我们对分片 2 也做同样的处理，得到区间 `["f","j")` 和 `["j","n")`。现在可以把 100 GB 数据（`["d","f")`）从分片 1 迁移到分片 4 上，把区间 `["j","n")` 内的所有文档从分片 2 迁移到分片 3 上（见图 2-6）。一个区间的数据称为一个数据块（也叫块，chunk）。当我们把一个块的区间一分为二时，一个块也变成了两个块。

这样每个分片上就有了 400 GB 数据，而仅有 200 GB 数据需要挪动。

如果要添加新分片，MongoDB 可以从每个分片顶端取 100 GB 数据并把这些块挪到新分片上，使得新分片获取 400 GB 数据需要移动的数据量最小化：只有 400 GB（见图 2-7）。

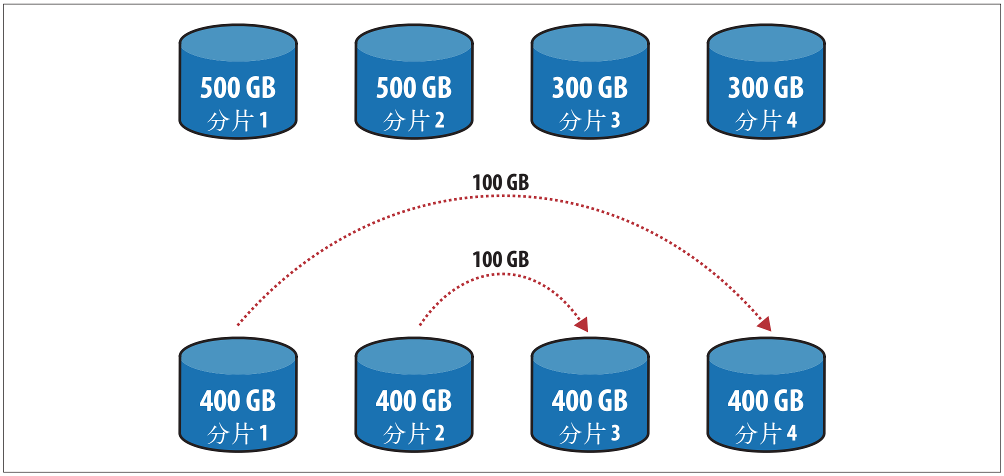


图 2-6：让分片支持多重非连续区间，我们便可以挑选数据并将其移动到任何地方

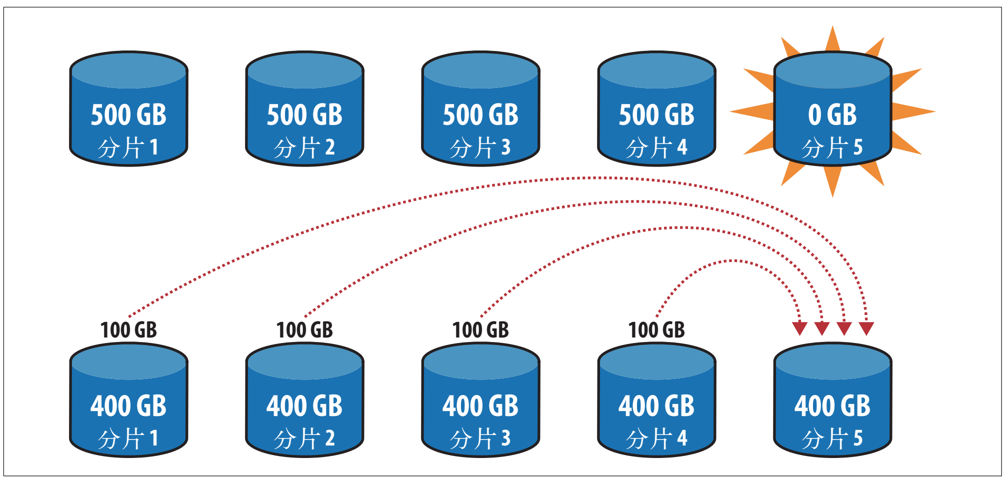


图 2-7：添加新分片时，每个分片都可以直接给它提供数据

这就是 MongoDB 的数据分配之道。当一个块变得越来越大时，MongoDB 会自动将其分割成两个较小的块。如果分片间比例失调，则 MongoDB 会通过迁移块来确保均衡。

2.1.2 如何创建块

当你决定分配数据时，必须为块区间选择一个键（前面我们一直在用 `username`）。这个键叫做*片键*（shard key），片键可以是任意字段或字段的组合。（第 3 章会介绍如何选择片键以及对集合进行分片的命令。）

1. 示例

假如我们的集合中包含如下文档（忽略 _id 字段）：

```
{"username" : "paul", "age" : 23}
{"username" : "simon", "age" : 17}
{"username" : "widdly", "age" : 16}
{"username" : "scuds", "age" : 95}
{"username" : "grill", "age" : 18}
{"username" : "flavored", "age" : 55}
{"username" : "bertango", "age" : 73}
{"username" : "wooster", "age" : 33}
```

如果我们选择 age 字段作为片键并得到一个块区间 [15, 26)，则此块会包含以下文档：

```
{"username" : "paul", "age" : 23}
{"username" : "simon", "age" : 17}
{"username" : "widdly", "age" : 16}
{"username" : "grill", "age" : 18}
```

如你所见，这个块中的所有文档其 age 字段值都在这个块的区间内。

2. 对集合分片

对一个集合分片时，无论集合里有什么数据 MongoDB 都只会创建一个块。这个块的区间是（- ∞ , ∞），其中 - ∞是 MongoDB 可以表示的最小值（也叫 $minKey），∞是最大值（也叫 $maxKey）。

> 如果被分片的集合中包含大量数据，MongoDB 会立刻把这个初始块分割为多个较小的块。

事实上，由于上面例子中的集合太小并不能触发分割，所以在插入更多数据之前，都只有一个块（- ∞ , ∞）。尽管如此，为了达到演示的目的，我们假设这个数据量已经足够大了。

MongoDB 会把初始块（- ∞ , ∞）分割为两个新块，分割位置一般选在已有数据区间的中点附近。因此，如果大约一半文档的 age 字段值小于 15，且另一半大于 15，MongoDB 就很可能会选择 15。这样就得到两个块：（- ∞ ,15] 和 [15, ∞ ）（见图 2-8）。如果我们往块 [15, ∞ ）里继续插入数据，它会再一次被分割（比如说分割成 [15,26）和 [26, ∞)）。这样集合中就有了 3 个块：（- ∞ ,15）、[15,26）和 [26, ∞)。在插入更多数据的同时，MongoDB 会持续地将已有块分割成更多的新块。

块的区间可以只包含一个值（例如仅包含用户名为 paul 的用户），但是各块的区间必须互不相同（不能有两个块的区间同为 ["*a*","*f*")）。另外，也不能有区间相互重

叠的块，而且每个块的区间必须紧邻下一个块的区间。因此如果要分割一个区间为 [4,8）的块，结果可以是 [4,6）和 [6,8）（因为二者合起来能覆盖原块的区间），但不可以是 [4,5）和 [6,8）（因为这样集合将丢失区间 [5,6）中的所有数据），也不能是 [4,6）和 [5,8）（因为会造成块的部分重叠）。每个文档必须属于且仅属于一个块。

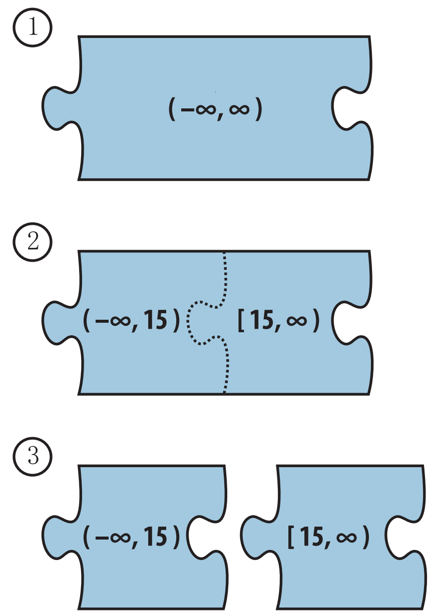


图 2-8：一个块分割成两个块

由于 MongoDB 不强制要求任何形式的结构定义，你可能会纳闷：那没有值可以作为片键的文档会被放到哪里去呢？实际上 MongoDB 并不允许插入无片键的文档（尽管使用 `null` 作为片键值是可以的），也不允许修改文档的片键值（例如用 `$set` 命令）。给文档一个新片键的唯一方法是先删除文档，然后在客户端修改片键的值，再重新插入文档。

如果在一些文档中使用字符串，而在另一些文档中使用数字呢？这也是可以的，因为在 MongoDB 中类型之间有严格的次序。如果在 `age` 字段插入一个字符串（或数组、布尔值、`null` 等），MongoDB 会按照类型对其排序。类型的先后次序如下：

`null` < 数字 < 字符串 < 对象 < 数组 < 二进制数据 < `ObjectId` < 布尔值 < 日期 < 正则表达式

在同一种类型内，排序与你所期望的很可能相同：2 < 4 或 `"a"<"z"`。

在前面的第一个例子里，块都是几百 GB 大小，但在真实系统中，块大小默认仅有 200 MB。这是因为挪动数据的代价非常大，要花费很长时间，占用系统资源，而且会明显增加网络流量。你可以自己试试看，先插入 200 MB 数据到某一集合中，然后试着取回所有 200 MB 数据。接着想象一下在建有多个索引（比如生产系统很可能有多个索引）的系统上做同样的事情，同时该系统上还有其他数据流量存在的情况。你肯定不会愿意看着应用程序逐渐降低性能直至停止，而 MongoDB 还在后台拖拖拉拉地挪动数据。实际上，如果一个块过大，MongoDB 根本就不会移动它。反过来你也不会希望块过小，因为每个块都需要有一点点管理开销（以便你不必为跟踪不计其数的数据块而烦恼）。综合考虑之下，我们发现 200 MB 恰好是兼顾可移动性和最小开销的最佳选择。

> 块是一个逻辑概念，而非物理实现。一个块中的文档在物理上并不连续存储于磁盘上或以任何形式聚集在一起。它们可能分散在整个集合的任何角落里。一个文档属于一个块当且仅当其片键值在对应的块区间内。

2.2 平衡

如果存在多个可用的分片，只要块的数量足够多，MongoDB 就会把数据迁移到其他分片上。这个迁移过程叫做*平衡*（balancing），由叫做*平衡器*（balancer）的进程负责执行。

平衡器会把数据块从一个分片挪到另一个分片上，其优点在于自动化，即你无需担心如何保持数据在分片间的均匀分布，这项工作已经由平衡器替你搞定了。不过这也是它的缺点，因为自动意味着如果你不喜欢它做负载均衡的方式，那只能算你不走运。如果不想让某个块存在于分片 3 上，你可以把它手动移动到分片 2 上，但是平衡器很可能把它再挪回分片 3。你只能选择要么对集合重新分片（re-shard），要么关闭平衡化。

在写这本书时，平衡器的算法还不是很智能。它每天基于分片整体大小来移动块。在（不久的）将来它会变得更加先进。

平衡器的目标不仅是要保持数据均匀分布，还要最小化被移动的数据量。因此触发平衡器需要很多条件。要触发一轮平衡，一个分片必须比块最少的分片多出至少 9 个块。到那时，块就会被迁移出拥挤的分片，直到与其他分片平衡为止。

平衡器并不非常激进的原因在于 MongoDB 希望能避免将相同数据来回移动。如果平衡器要平衡掉每一点微小的差别，那很可能会不停地浪费资源：分片 1 比分片 2

多两个块时，它就发送一个块给分片 2。接着一些写操作落到分片 2 上，使得分片 2 又比分片 1 多出两个块，结果同一块数据又被折腾回去（见图 2-9）。通过等待更严重的不平衡发生，MongoDB 能够最小化无意义的数据传输。要知道差 9 个块其实也不是那么不平衡，因为这还不到 2 GB 数据呢。

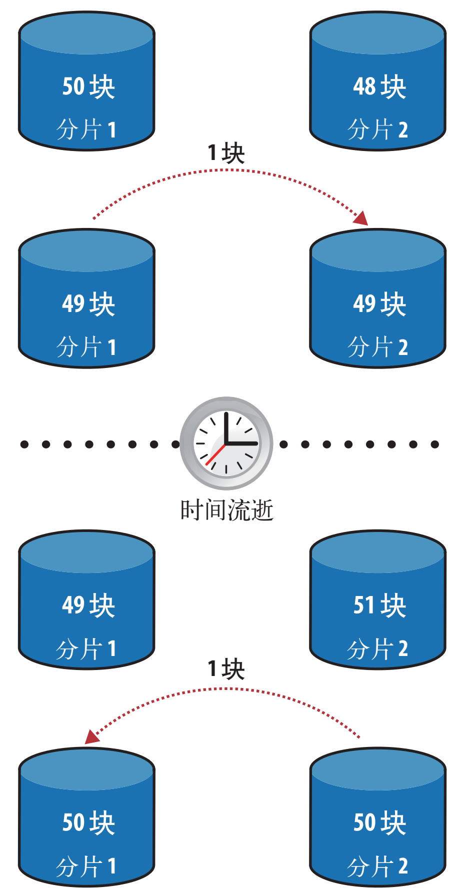


图 2-9：如果每一点轻微的不平衡都被修正，则最终必然导致大量不必要的数据移动

3. 每日平衡背后的病理心理学

大部分用户希望通过看到数据移动来向自己证明分片可以工作，这产生了一个问题：触发一轮平衡所需的数据量远比大多数人想象的大。

比方说我正在尝试分片，于是写了一个命令行脚本来插入 50 万份文档到分片集合中：

```
> for (i=0; i<500000; i++) {
    db.foo.insert({"_id" : i, "x" : 1,"y" : 2, "z" : i, "date" : new Date(),
      "foo" : "bar"});
}
```

等插入完成，我应该能看见一些数据飞来飞去了，对不对？错。如果我看一眼数据库状态，就会发现还差得远呢（为了清楚省略了一些字段）：

```
> db.stats()
{
    "raw" : {
        "ubuntu:27017" : {
            /* 分片状态 */
        },
        "ubuntu:27018" : {
            /* 分片状态 */
        }
    },
    "objects" : 500008,
    "avgObjSize" : 79.99937600998383,
    "dataSize" : 40000328,
    "storageSize" : 69082624,
    "ok" : 1
}
```

如果你看一眼 `dataSize`，可以发现有 40 000 328 字节的数据，大致等于 40 MB。这还不够一个块。甚至还不够一个块的 1/4！真要想看到数据移动的话，需要插入 2 GB 数据，也就是 2500 万个这样的文档，或者说是现在已插入数据的 50 倍。

开始使用分片时，人们希望看到数据到处移动，这是人的本性使然。尽管如此，在生产系统中你不会希望发生太多迁移，因为这种操作代价极其高昂。因此，一方面我们希望看到迁移实际发生，而另一方面，事实又是如果它不是看上去慢得让人烦躁，就不可能工作得很好。

对这个矛盾，MongoDB 需要实现两个“技巧”，让分片更加善解人意，同时又不对生产系统造成破坏性影响。

技巧 1：自定义块大小

第一次启动 *mongos* 时，可以声明 `--chunkSize N` 参数，其中 `N` 就是你想要的块大小，单位是 MB。如果只是想试试分片，可以设置 `--chunkSize 1`，这样只要插入几 MB 的数据就能看到迁移发生。

技巧 2：递增块大小

即使是部署一个真正的应用程序，要达到 2 GB 看起来也遥遥无期。所以对于前十几

个块，MongoDB 会特意自动降低块大小，从 200 MB 降到 64 MB，而这就是为了更好地照顾一下用户的感受。一旦数据块多起来，它会自动把块大小递增回 200 MB。

改变块大小

块大小可以通过在启动时指定 `--chunkSize N` 参数或修改 *config.settings* 集合并整体性重启来改变。尽管如此，除非是为了好玩试试 1 MB 的块大小，请勿随意改动块大小。

我可以保证，无论你正在尝试解决什么问题，摆弄块大小绝不是根本的解决之道。它看上去很诱人，但请勿这么做。除非你只是想设置 `--chunkSize 1` 试试，否则请保持块大小不变。

2.3 mongos

mongos 是用户和集群间的交互点，其职责是隐藏分片内部的复杂性并向用户（你）提供一个简洁的单服务器接口。这个抽象层中也存在一些“缝隙”（参见第 4 章），不过大多数情况下 *mongos* 允许你把一个集群当作一台服务器。

使用集群时，应该连接一个 *mongos* 并向它发送所有的读写操作。无论如何，你都不应该直接访问分片（但如果想的话能做到）。

mongos 会将所有用户请求转发到恰当的分片上。如果用户插入一份文档，*mongos* 会查看文档的片键，对照数据块，并把文档发送到持有相应块的分片上。

举个例子，比如说我们要插入 `{"foo":"bar"}` 且已经以 `foo` 作为片键做了分片。*mongos* 会查看所有可用的块，然后发现有一个区间为 `["a","c")` 的块应当包含 `"bar"`。这个块在分片 2 上，因此 *mongos* 会把插入消息发送给分片 2（见图 2-10）。

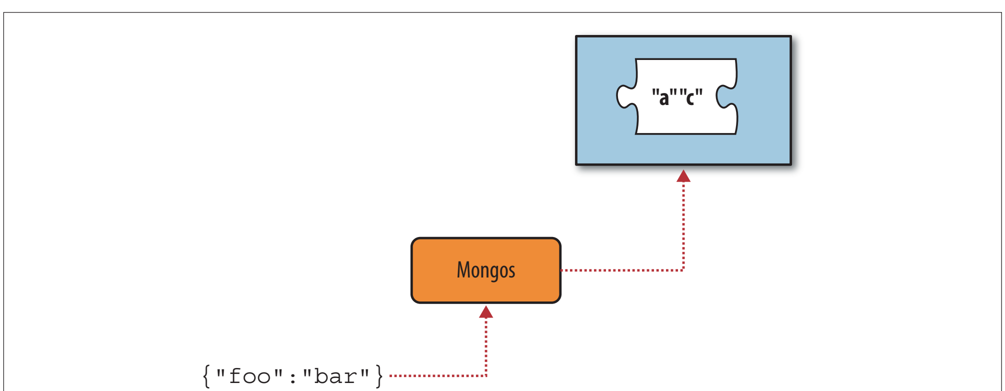


图 2-10：把一个请求路由到包含对应块的分片上

如果一个查询涉及了片键，*mongos* 就可以使用和插入一样的流程并找到某个（或某些）正确的分片并向其发送查询。这种查询又称为针对性查询（targeted query），因为它只针对那些可能包含我们所要查找的数据的分片。如果它知道我们在查找 `{"foo":"bar"}`，那么查询所含片键值大于 `"bar"` 的分片就没有什么意义了。

如果查询不包含片键，*mongos* 就必须把查询发送给所有分片。这可能会比针对性查询效率低，但并不一定。如果一个泛泛的查询仅访问内存中少量已被索引的文档，而一个针对性查询不得不访问分布在多个（甚至所有）分片中的磁盘数据，两相比较，前者的性能肯定要比后者高得多。

2.4 配置服务器

mongos 进程并不持久地存储任何数据，集群配置被保存在一组专门的 *mongod* 上，它们被称作配置服务器（config server）。配置服务器包含了有关集群的最完整可靠的信息以供所有人（分片、*mongos* 进程和系统管理员）访问。

要保证数据块迁移成功，所有配置服务器都必须同时在线。如果其中任意一台停机了，则当前正在执行的所有迁移都会回退并停止，直到整套配置服务器再一次运行起来。任何一台配置服务器停机，集群配置都无法改变。

2.5 集群的构造

一个 MongoDB 集群基本上由 3 类进程组成，即实际存储数据的分片、负责把请求路由到正确数据的 *mongos* 进程，以及用于跟踪集群状态的配置服务器（见图 2-11 和图 2-12）。

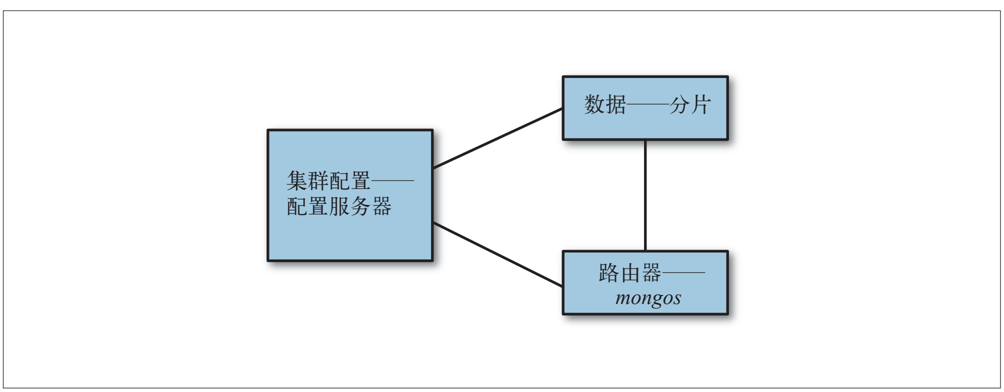


图 2-11：一个集群的 3 个组件

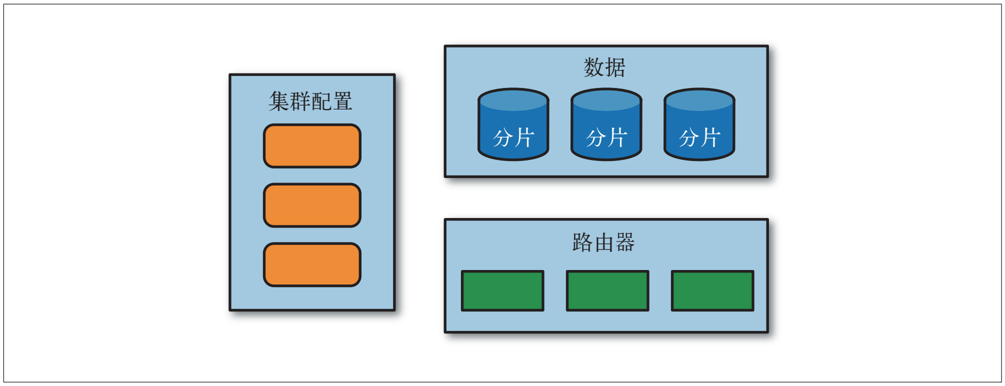


图 2-12：每个组件可以包含多个进程

以上每个组件都不是“一台机器”。*mongos* 进程通常跑在应用程序服务器（appserver）上。配置服务器，鉴于其重要性，是非常轻量级的并且基本上可以在任何机器上运行。每个分片通常由多台机器组成，因为它们实际存储数据。

第 3 章

建立集群

3.1 选择片键

选择一个好的片键非常关键。如果选择了一个糟糕的片键，它可以立马或者在访问量变大时毁了你的应用程序，也有可能潜伏着，等待着，没准儿什么时候突然毁了你的应用程序。

另一方面，如果你选择了一个好片键，只要应用程序还在正常运行，而且只要发现访问量提高就赶紧添服务器，MongoDB 就会确保一直正确地运行下去。

正如在上一章中所学的，片键决定了数据在集群中的分布情况。因此你会希望存在这样一个片键，它既能把读写分散开来，又能把正在使用的数据保持在一起。这些看似相互矛盾的目标在现实中却往往是可以实现的。

我们先挑片键的几个反面例子找找茬，然后再拿几个较好的例子来琢磨一番。MongoDB 的 wiki 上也有一页与选择片键相关且很不错的内容，可以看看。

3.1.1 小基数片键

一些人并不真正理解或者信任 MongoDB 自动分配数据的方式，所以他们总是沿着这么个思路来想："我有 4 个分片，所以应该用一个有 4 个可能值的字段来做片键。"这是一个非常糟糕的想法。

为什么呢？

假设我们有一个存储用户信息的应用程序。每个文档有一个 `continent` 字段代表用户的所在地区，其字段值可以是 `"Africa"`、`"Antarctica"`、`"Asia"`、`"Australia"`、`"Europe"`、`"North America"` 或 `"South America"`。考虑到我们在每个大洲都有一个数据中心——或许不包括南极洲（Antarctica）——并且想从人们所在"当地"的数据中心为其提供用户数据，我们决定按该字段进行分片。

集合开始于某个数据中心的一个分片的初始块（区间 (-∞, ∞)），所有的插入和读取都落在这一个块上。一旦它变得足够大，就会被划分成两个块（区间分别为 `(-∞,"Europe")` 和 `["Europe",∞)`）。这样一来，所有来自非洲（Africa）、南极洲（Antarctica）、亚洲（Asia）或澳大利亚（Australia）的文档都会被分到第一块上，而所有来自欧洲（Europe）、北美（North America）或南美（South America）的数据都会被分到第二块上。随着更多文档被添加进来，集合最终会变成 7 个块（见图 3-1）：

- `(-∞,"Antarctica")`
- `["Antarctica","Asia")`

- ["Asia","Australia")
- ["Australia","Europe")
- ["Europe","North America")
- ["North America","South America")
- ["South America", ∞)

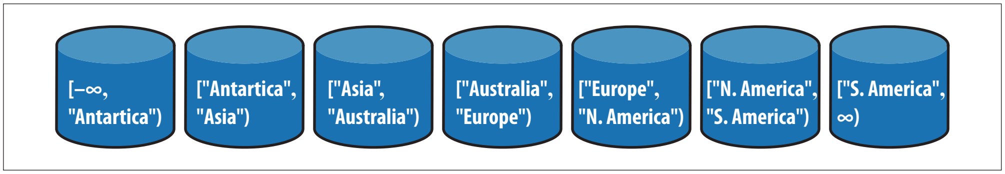


图 3-1：每个大洲一个分片

然后呢？

MongoDB 不能再进一步分割这些块了！块只能变得越来越大。虽然暂时不会出问题，但是当服务器磁盘空间开始逐渐耗尽时问题就会浮出水面了。除了买块更大的磁盘，你什么也做不了。

由于片键值数量有限，因此这种片键称为*小基数片键*（low-cardinality shard key）。如果选择了一个基数很小的片键，到头来肯定会得到一堆既巨大又无法移动，还不能分割的块，它们会让你极为不快，这样的生活你懂的。

如果这么做是为了手动分配数据，那就不要再用 MongoDB 内置的分片机制了，否则它会和你一直抗争到底。当然，不管怎样你还是可以手动对集合进行分片，编写自己的路由器，并将读写路由到任何一台（组）服务器上。只不过选择一个好片键并让 MongoDB 来替你做这一切更容易一些。

1. 适用的键

这个规则适用于任何取值个数有限的键。一定要记得，如果在某集合中一个键有 *N* 个值，那就只能有 *N* 个数据块，因此也只能有 *N* 个分片。

如果打算采用小基数片键的原因是需要在那个字段上进行大量查询，请使用组合片键（一个片键包含两个字段），并确保第二个字段有非常多不同的值可以供 MongoDB 用来进行分割。

2. 例外

如果一个集合有生命周期（例如，每周都创建一个新的集合而且你知道，在一周时间里数据量不会接近任何分片的最大容量），就可以选择这个生命周期为片键。

3. 数据中心感知

这个例子不仅仅是关于选择小基数片键的，还与在 MongoDB 分片机制中添加*数据中心感知*（data-center awareness）支持的尝试有关。到目前为止，分片尚不支持数据中心感知。如果对此感兴趣，可以查看或对相关的问题投票。[1]

靠使用者自己实现的问题在于其扩展性并不是很好。如果你的应用最近在日本很流行怎么办？你可能会想添加第二个分片来应付亚洲地区的访问量。

但是你打算如何迁移数据呢？一个数据块增长到好几 GB 大，你就没法移动它了，而且也不能再分割块，因为整个块就只有一个片键值。由于片键值无法更新，因此也不可能通过更新所有文档来使用一个更独特的片键值。可以删除每个文档，更新片键值，然后再把它重新保存回去，但是对于大型数据库而言那并不是一个能迅速完成的操作。

你所能做的最好的事情就是在插入文档时开始用 `"Asia,Japan"` 替代简单的 `"Asia"`。这样一来会有一批旧文档，其片键值应该是 `"Asia,Japan"` 但却是 `"Asia"`，因此应用程序逻辑就不得不同时支持两种情况。另外，一旦开始拥有更细粒度的块，就不能保证 MongoDB 还会把它们放在你所期望的地方（除非关闭平衡器并手动处理所有事情）。

数据中心感知对于大型应用非常重要，而且它对 MongoDB 的开发者有很高的优先级。在这期间选择一个小基数片键绝不会是一个好的解决方案。

3.1.2　升序片键

从 RAM 中读取数据要比从磁盘中读取快，所以目标是尽可能多地访问内存中的数据。因此，如果有些数据总是被一起访问，我们就希望片键能够把它们保持到一起。对大部分应用程序而言，新数据被访问的次数总比老数据多，所以人们往往会尝试使用诸如时间戳或者 `ObjectId` 一类的字段来做片键。但是这并不像他们所期望的那样可行。

比方说我们有一个类似微博的服务，其中每个文档都包含一条短消息、发送人以及发送时间。我们按发送时间字段来分片，取值为自公元元年起经过的秒数。

和往常一样，还是从一个数据块（- ∞ , ∞）开始。全部插入都会落到这个分片上直至它分裂成两个块，比如说（- ∞ , 1294516901）和 [1294516901, ∞）。由于是从片键中点把块分开来的，所以在我们分割块的那一刻，时间戳很可能已经远大于 1294516901。这意味着再往后所有的插入都会落到第二个块上，不会再有插入操作

译注1：2.0版本已发生变化，详情见http://www.mongodb.org/display/DOCS/2.0+Release+Notes#2.0ReleaseNotes-Datacenterawareness。

命中第一个块。一旦第二个块填满了，它就会分裂成 [1294516901,1294930163）和 [1294930163, ∞）两个块。但是因为从现在起时间都在 1294930163 之后，所以新的插入都会被添加到区间为 [1294930163, ∞）的块上。这个模式会持续下去：所有数据总是被添加到“最后”一个数据块上，即所有数据都会被添加到一个分片上。

这个片键创造了一个单一且不可分散的热点。

1. 适用的键

这条规则适用于任何升序排列的键值，而并不必须是时间戳。其他例子包括 `ObjectId`、日期、自增主键（很可能是从其他数据库导入的）。只要键值趋向于无穷大，你就会面临这个问题。

2. 例外

基本上，这种片键总是一个坏主意，因为它导致热点必然存在。如果访问量不大且用一个分片就能承受所有读写，那还行得通。当然，如果遇到一个访问量尖峰或者应用开始变得更受欢迎，那它终会停止工作并且难以修复。

除非你非常清楚自己在干什么，否则不要使用升序片键。肯定还有更好的片键存在，应该避免使用这一个。

3.1.3　随机片键

有时为了避免热点，人们会选择一个取值随机的字段来分片。采用这种片键一开始还不错，但是随着数据量越变越大，它会变得越来越慢。

假设我们在分片集合中存储照片的缩略图。每个文档都包含了照片的二进制数据、二进制数据的 MD5 散列值，以及一段描述、拍照时间和拍照人。我们决定在 MD5 散列值上做分片。

随着集合的增长，我们最终会得到一组均匀分布于各分片的数据块。目前为止一切顺利。现在，假设我们非常忙而分片 2 上的一个块填满并分裂了。配置服务器注意到分片 2 比分片 1 多出了 10 个块并判定应该抹平分片间的差距。这样 MongoDB 就需要随机加载 5 个块的数据到内存中并将其发送给分片 1。考虑到数据序列的随机性，一般情况下这些数据可能不会出现在内存中。所以此时的 MongoDB 会给 RAM 带来更大压力，而且还会引发大量磁盘 IO（磁盘 IO 总是非常慢）。

除此以外，片键上必须有索引，因此如果选择了从不依据它进行查询的随机键，基本上可以说是“浪费”了一个索引。另外，考虑到每增加一个索引都会让写操作变得更慢，所以保持索引数量尽可能低也是非常重要的。

3.1.4 好片键

我们真正需要的是一种将访问模式也考虑进去的方案。如果应用会规律性地访问 25 GB 的数据，我们就希望所有的分割和迁移都发生在这 25 GB 数据上，而不是随机访问数据以至于不断地有新数据被从磁盘中复制到内存里。

因此我们希望能找到这样一个片键，它具备有良好的*数据局部性*（data locality）特征，但又不会因太局部而导致热点出现。

1. 准升序键加搜索键

许多应用访问新数据比老数据更频繁，所以我们希望数据大致上按照时间排序，但是同时也要均匀分布。这样一来既能把我们正在读写的数据保持在内存中，又可以使负载均衡地分散在集群中。

我们可以通过像 `{coarselyAscending:1,search:1}` 这样的*组合片键*（compound shard key）来实现目标。其中 `coarselyAscending` 键的每个值最好能对应几十到几百个数据块，而 `search` 键则应当是应用程序通常都会依据其进行查询的字段。

举个例子，比如说有个分析程序，用户会定期通过它访问过去一个月的数据，而我们希望能尽量保持数据易于使用。因此可以在 `{month:1,user:1}` 上分片，其中 `month` 是一个粗粒度的升序字段，即每个月它都会有一个更新更大的值。`user` 适合作为第二个字段，因为我们会经常查询某个特定用户的数据。

还是从一个数据块开始，组合区间是 ((- ∞ , - ∞) , (∞ , ∞))。当它被填满，我们将其分为两个块，比如区间 ((- ∞ , - ∞) , ("2011-04","susan")) 和 [("2011-04","susan") , (∞ , ∞))。假设现在还是 4 月份（"2011-04"），则所有的写操作都会被均匀地分到两个数据块上。所有用户名小于 "susan" 的用户都会被放在第一个块上，而所有用户名大于 "susan" 的用户都会被放在第二个块上。

随着数据持续增长，这个方案还会继续有效。4 月里后续创建的块都会被移动到不同的分片上，从而确保了读写在集群中的负载均衡。等 5 月到来时，我们开始创建边界为 "2011-05" 的块。随着 6 月悄然而至，"2011-04" 的数据块已然无人问津，所以就可以将这些块安静地换出内存使之不再占用资源。尽管以后仍有可能因为历史原因再查看这些块，但是应该不需要再分割或迁移它们了（使用随机索引难以避免的问题）。

2. 常见问题

为什么不将 `{ascendingKey : 1}` 用作片键？

在 3.1.2 节提到过，如果你把它和一个粗粒度的升序键合在一起用，还是会创造出一堆巨大且无法分割的数据块。

search 字段可不可以也是升序字段？

不可以。如果是，则该片键会降级成一个升序片键，进而使你同样面临普通升序键带来的热点问题。

search 字段应该是什么？

search 字段最好是应用程序可以用于查询的东西，比如用户信息（比如上面的例子）、文件名字段或者 GUID 等。它应该具备非升序、分布随机且基数适当的特点。

3. 一般情况

基于上面的内容，我们可以概括出一个片键公式：

```
{coarseLocality : 1,search : 1}
```

其中 coarseLocality 字段用来控制数据局部化。search 字段则是数据上常用到的一个检索字段。

这个键并不是唯一可能的片键，也不会在任何情况下都有效。尽管如此，即便你最终并不使用它，但就启发思考如何选择片键而言，这仍是一个好的开始。

4. 我该用哪种片键

真的没办法告诉你，因为我并不了解你的应用程序。不过选择一个好的片键应该会花费一些功夫。在选定一个片键之前，不妨先想想下面这些问题的答案。

- 写操作是怎样的？你要插入什么文档，有多大？
- 系统每小时会写多少数据？每天呢？高峰期呢？
- 哪些字段取值是随机的，哪些是增长的？
- 读操作是怎样的？人们在访问哪些数据？
- 系统每小时会读多少数据？每天呢？高峰期呢？
- 数据做索引了吗？应不应该索引呢？
- 数据总量有多少？

也许你还会在数据中发现其他一些模式，那就利用起来！在进行分片之前，你需要清楚地了解你的数据。

3.2　新老集合分片

一旦选择好片键，就可以进行数据分片了。

3.2.1　快速起步

如果想尽快建立起一个集群玩玩看，用 Github 上的 mongo-snippets 只要一两分钟就能建好一个。它的文档有点少，但是这个库基本上就是一组对用户非常有用的脚本。其中特别有趣的一个是 *simple-setup.py*，它能自动地启动、配置和生成一个集群（本地的）。

要运行这个脚本，需要 MongoDB 的 Python 驱动。如果没有，可以通过执行下面的语句来安装（在 *NIX 系统上）：

```
$ sudo easy_install pymongo
```

一旦 Pymongo 装好了，下载 *mongo-snippets* 库并执行下面的命令：

```
$ python sharding/simple-setup.py --path=/path/to/your/mongodb/binaries
```

这里有很多其他可用的选项，执行 `python sharding/simple-setup.py-help` 可查阅它们。这个脚本有点挑剔，所以请确保使用完全路径（例如 /home/user/mongo-install，而非 *~/mongo-install*）。

simple-setup.py 会启动一个 *mongos* 进程，地址为 *localhost*:27017，这样就可以用 shell 命令连上它来使用了。

如果想建立一个企业级集群，请继续读下去。

3.2.2　配置服务器

你可以运行 1 个或 3 个配置服务器。我们会采用 3 个配置服务器，在生产环境里任何时候都应当这么做。

> "1 个或 3 个"是个奇怪且有些专断的限制，原因是 1 个适于测试环境，而 3 个适于生产环境。2 个对测试环境来说多了，对生产环境来说又不够。
>
> 不能运行任意个数的配置服务器，因为它们之间的交互是复杂的，而且 *M* 个配置服务器中停掉 *N* 个时的处理逻辑和编程都非常困难。将来 MongoDB 可能会支持任意个数的配置服务器，但这并不是当务之急。

设置配置服务器很乏味，因为它们只是普通的 *mongod* 实例。设置时唯一要注意的是，由于需要它们中的部分配置服务器一直正常运行，所以应尽可能把它们分到不

同的故障域（failure domain）里。

比方说我们有 3 个数据中心，一个在纽约，一个在旧金山，还有一个在月球上。此时只需每个数据中心启动一个配置服务器。

```
$ ssh ny-01
ny-01$ mongod

$ ssh sf-01
sf-01$ mongod

$ ssh moon-01
moon-01$ mongod
```

就这么简单！

你可能会从上面的设置中发现一个小问题。这些本应联系密切的服务器并不知道其他服务器的存在抑或知道它们扮演着配置服务器的角色。没有问题——接下来我们会介绍它们相互认识。

> 有时人们认为必须在配置服务器上设置复制。实际上配置服务器并不使用“普通”*mongod* 所采用的复制机制，而且也不应该按副本集或主从设置启动。只要把配置服务器当做未连接的普通 *mongod* 启动就行了。

当你启动配置服务器时，运行 `mongod -help` 的输出中“General Option”下面的选项可以随意使用，但 `--keyFile` 和 `--auth` 除外。分片现在还不支持认证，但在到达 1.9 版的半路上这应该有所改变。[1]

带 `--quiet` 参数运行可不是一个好主意，因为如果出了问题就需要能弄清楚发生了什么。可以用 `--logpath <file>` 和 `--logappend` 来代替 `--quiet` 将日志发送到某处，这样如果出问题，就有日志可以回溯。

Sharding（分片）选项适用于分片，而非配置服务器，所以可以忽略它们。你也不需要其他（Replication、Replica set 或者 Master/slave 下面的）选项，因为你不会在配置服务器上启用复制机制，对吧？

3.2.3 mongos

接下来是启动 *mongos* 进程。一个分片配置需要至少一个 *mongos*，但无上限。要记住必须（至少应该）监视所有 *mongos* 进程，因此你很可能不会想启动数千个进程，

译注1：2.0版本已发生变化，详情见http://www.mongodb.org/display/DOCS/2.0+Release+Notes#2.0ReleaseNotes-ShardingAuthentication。

一般来说每个应用程序服务器一个 *mongos* 进程就挺好。接下来我们登录应用程序服务器并启动第一个 *mongos*。

```
$ ssh ny-02
ny-01$ mongos --configdb ny-01,sf-01,moon-01
```

按下回车，现在所有配置服务器都知道其他配置服务器了。*mongos* 就像派对上的主人，会介绍宾客们相互认识。

现在你有了一个微型集群。尽管还不能存任何数据进去（配置服务器只存配置不存数据），但是除此以外，它完全可以正常工作。

你可能想用数据库读写数据，所以让我们添一些数据存储机制进来。

3.2.4 分片

集群上的所有管理工作都是通过 *mongos* 完成的。所以，先用 shell 命令连上之前启动的 *mongos*。

```
$ mongo ny-02:27017/admin
MongoDB shell version: 1.7.5
connecting to: admin
>
```

确认当前使用的是 *admin* 数据库，设置分片需要在 *admin* 数据库中执行命令。

一旦连上，就可以添加分片了。添加分片有两种方法，使用哪种取决于分片是单个服务器还是副本集。假设我们有一个服务器 *sf-02*，一直用来放数据。执行 `addShard` 命令我们可以让它成为第一个分片：

```
> db.runCommand({"addShard" : "sf-02:27017"})
{ "shardAdded" : "shard0000", "ok" : 1 }
```

这个命令将服务器添加到集群中。现在可以存储和查询数据了。（如果存储数据时还不存在任何可以用于存储数据的分片，MongoDB 会提示错误，理由不言自明。）

通常情况下，应该用副本集（而非单个服务器）来做每个分片。副本集能提供更好的可用性。要添一个副本集分片，必须传给命令 `addShard` 一个形如 `"setName/seed1[,seed2[,seed3[,...]]]"` 的字符串，即必须提供副本集的名称和至少一个集合成员（*mongos* 可以推出其他成员，前提是它能连上它们中任意一个）。

举个例子，如果我们有个副本集叫做 rs，其成员为 *rs1-a*、*rs1-b* 和 *rs1-c*。我们可以执行：

```
> db.runCommand({"addShard" : "rs/rs1-a,rs1-c"})
{ "shardAdded" : "rs", "ok" : 1 }
```

注意，这一次得到的名称漂亮多了！（我就是那个写程序让它挑出副本集名称的人，所以对这点感到特别自豪。）如果添加一个副本集，则它的名称会成为分片的名称。

> 可以随意命名分片。如果不想使用默认值，就在添加分片时加上 name 字段。
>
> ```
> > db.runCommand({"addShard" : "sf-02:27017", "name" :
> "Golden Gate shard"})
> { "shardAdded" : "Golden Gate shard", "ok" : 1 }
> > db.runCommand({"addShard" : "set1/rs1-a,rs1-b", "name" :
> "replicaSet1"})
> { "shardAdded" : "replicaSet1", "ok" : 1 }
> ```
>
> 某些场合下你将不得不使用名字来引用分片（见第 5 章），所以别把名字定得太长太疯狂。

限制分片大小

默认情况下，MongoDB 会在分片间均匀地分配数据。如果使用的是一组通用配置服务器，那这就很有用，但如果有一个台 10 TB 的剽悍机器，而另一台是就几百 GB 的普通机器，那就有问题了。如果服务器失衡严重，应该使用 maxSize 选项，它可以用来指定分片能增长到的最大存储量，单位 MB。

要记住 maxSize 更像是一个建议而非规定。MongoDB 不会从 maxSize 处截断一个分片，或是阻止它再增加哪怕一个字节，而是会停止将数据移动到该分片，当然还可能会挪走一部分数据，如果它觉得这么做合适的话。所以设置这个值时可以略微低一些。

maxSize 是 addShard 命令的又一个选项，如果想要设置一个分片只使用 20 GB，可以执行：

```
> db.runCommand({"addShard" : "sfo/server1,server2", "maxSize" : 20000})
```

将来，MongoDB 会自动分析出在每个分片上可以配置多少空间并相应地做计划。在此期间，你可以使用 maxSize 给它一个提示。

现在你终于有了一个全副武装且可以运行的集群！

3.2.5 数据库和集合

如果想要 MongoDB 来帮你分配数据，就要让它知道具体需要处理的数据库和集合。在尝试对一个数据库中的集合进行分片前，你必须先告诉 MongoDB 该数据库可以

包含分片集合。

假设我们在写一个博客程序，所有集合都在 *blog* 数据库中。我们可以用如下命令来启用分片：

```
> db.adminCommand({"enableSharding" : "blog"})
{ "ok" : 1 }
```

现在就可以对集合进行分片了。要进行分片必须指定集合和片键。假设我们在 `{"date" : 1,"author" : 1}` 上分片：

```
> db.adminCommand({"shardCollection" : "blog.posts", key : {"date" : 1,
"author" : 1}}
{ "collectionSharded" : "blog.posts", "ok" : 1 }
```

注意，命令里包含了数据库名 *blog.posts*，而非 *posts*。

如果要对一个已经包含数据的集合进行分片，数据片键上必须有索引。所有文档也都必须有片键值（且值不能为 `null`）。在集合分片完成后，才可以插入片键值为 `null` 的文档。

3.3 增减容量

随着应用持续增长，你需要添加更多的分片。添完新分片以后，MongoDB 就会从老分片向新分片移动数据。注意，数据移则压力增。当然，MongoDB 会尽可能轻柔地迁移数据，它会每次一个块地慢慢移动数据，如果服务器看起来很忙就等会儿再试。尽管如此，无论 MongoDB 多么轻巧地操作，移动数据块仍然会加重负载。

这意味着如果等服务器容量耗尽再添加分片，添加新分片可能会导致应用程序在一系列连锁反应下逐步瘫痪。假设现有分片就快要达到最大容量了，所以你添一个新分片进来帮助处理负载。平衡器会告诉分片 1 给新分片发送一个数据块。分片 1 的内存刚刚够容下应用程序运行时数据，而现在它将不得不把一整个数据块加载进内存。为了给新加载的块腾出足够空间，应用程序所使用的数据开始被换出 RAM。这意味着应用程序不得不开始访问磁盘，而 MongoDB 正在为那块被加载的数据访问磁盘。查询耗时开始延长，请求也逐渐排起长队，这进一步导致了问题恶化（见图 3-2）。

学到的教训是什么？要在还有足够空间做调度时添加分片。如果已经没有那么多空间了，就等到半夜三更（或者另外一个非忙时段）再添加分片。如果没有非忙时段，而且已经等了太长时间，还有一些办法来破解困局，但是这些方法既不有趣也不容易。（可以用现存备份让分片借尸还魂，然后手动更改 *config.chunks* 集合，再重启整个集群，但很明显这种做法并不被提倡。）所以添加分片要尽早做，经常做。

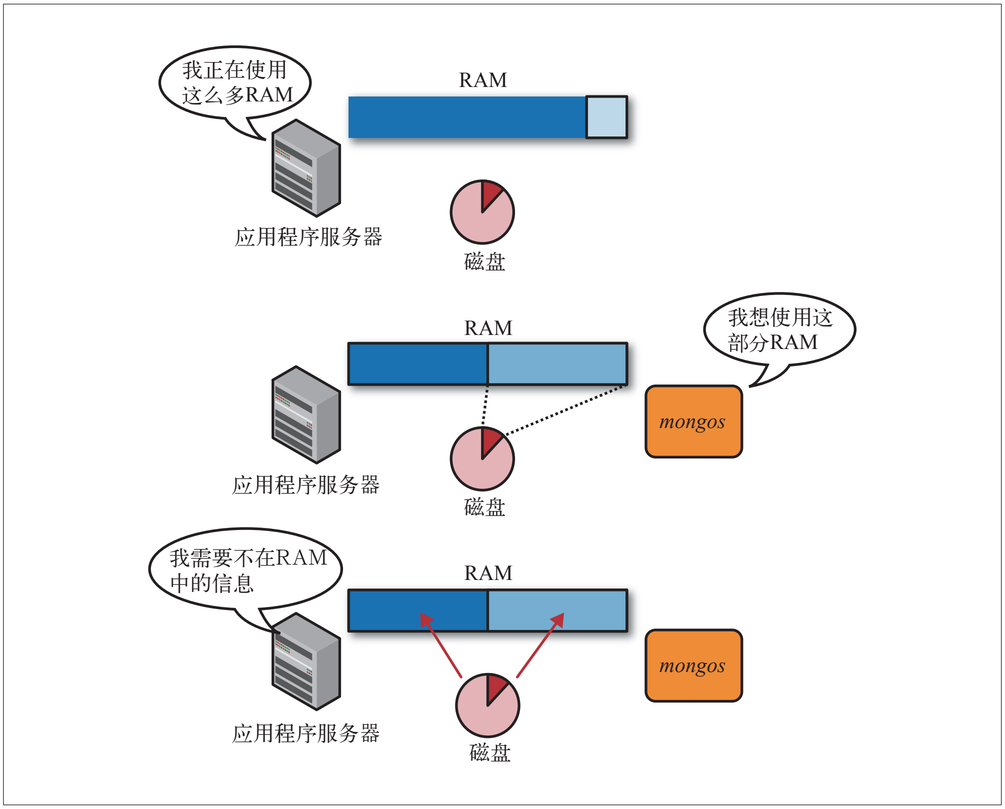


图 3-2：移动块会迫使数据被换出 RAM，导致更多请求命中磁盘并拖慢整个系统

那么再回想一下，如何添加分片呢？方法和前面添加第一个分片是一样的，用 addShard 命令。

之后再添加分片有一点很有意思，即分片不必是空的。分片不能包含集群中已有的数据库，但是如果集群里有一个 foo 数据库，而新添加的分片里包含 bar 数据库，那么不会产生任何问题，因为配置服务器会注意到它，而它会出现在集群信息中。因此，如果你愿意，可以把多个不同的系统集中起来攒出一个大号的分片集群。

3.3.1 移除分片

有时也需要移除分片。某些人可能过早地进行了分片或者选错了片键，所以想把所有东西都放回到一个分片上，然后转储，然后恢复，然后重新来过。你可能只是想让某些服务器下线。

removeShard 命令可以用于从集群中摘除分片。

```
> db.runCommand({removeShard : "Golden Gate shard"})
{
    "msg" : "draining started successfully",
    "state" : "started",
    "shard" : "Golden Gate shard",
    "ok" : 1
}
```

注意，消息是说“started successfully”（已启动成功）。在移除一个分片前需要把分片上的所有信息都转移到其他分片上。正如前面提到的，在分片间挪动数据非常耗时。removeShard 命令会立即返回，你必须通过轮询了解操作是否已经完成。如果再调用一次，就会看到像这样的消息：

```
> db.runCommand({removeShard : "Golden Gate shard"})
{
    "msg" : "draining ongoing",
    "state" : "ongoing",
    "remaining" : {
        "chunks" : NumberLong(2),
        "dbs" : NumberLong(1)
    },
    "ok" : 1
}
```

一旦完成，其状态会变成“completed”。那之后，就可以安全地关闭分片了（或者把它当做一个未分片的 MongoDB 服务器来用）。

```
> db.runCommand({removeShard : "Golden Gate shard"})
    {
    "msg" : "removeshard completed successfully",
    "state" : "completed",
    "shard" : "Golden Gate shard",
    "ok" : 1
}
```

这意味着分片已经完全清空并可以关机或挪做它用了。

3.3.2 修改分片中的服务器

如果使用了副本集，可以添加或删除副本集中的服务器，而 *mongos* 会注意到这些变化。要修改某个集群中的副本集，就假定它是独立运行的来进行：连接主服务器（而非通过 *mongos*）并修改相应配置。

第 4 章

使用集群

查询 MongoDB 集群通常和查询 *mongod* 是一样的。尽管如此，还是有些例外情况值得了解一下。

4.1 查询

如果使用副本集且 *mongos* 为 1.7.4 或更高版本，你可以把读分散到集群的从机上。这样就能很轻松地处理读负载，当然 CAP 原则也会应验：你必须要接受过时的数据。

要通过 *mongos* 查询一台从机，无论使用哪种驱动都必须设置“slave okay”选项（基本上就是确认接受有可能过期的数据）。在 shell 里，看起来就像这样：

```
> db.getMongo().setSlaveOk()
```

接着正常查询 *mongos* 即可。

4.2 为什么会这样

使用集群时，就不能把整个集合看做是一个“即时快照”（snapshot in time）了。很多人撞了南墙才意识到这个问题的严重性，所以让我们来看看这个问题对应用程序的一些常见影响。

4.2.1 计数

当你在一个分片集合上进行计数（count）时，可能得不到期望的结果，可能是一个比实际文档量多得多的数。

计数的工作方式是通过 *mongos* 将 `count` 命令发送给集群中的每个分片。然后，每个分片各自执行 `count` 并把结果返回给 *mongos*，*mongos* 再把结果累加起来最后返回给用户。如果有一个迁移正在进行，许多文档就会同时存在（进而被纳入计数）于多个分片上。

当 MongoDB 迁移数据块时，首先将块从一个分片复制到另一个分片。期间所有对该块的读写操作仍然会被路由到原分片，不过块会在另一边的分片上被逐渐填充。一旦数据块“移动”完成，它实际上存在于两个分片。最后一步，MongoDB 才会更新配置服务器并删除原来分片上的数据副本（见图 4-1）。

因此，当这部分数据被计数时，实际上被算了两次。将来 MongoDB 也许会解决这个问题，但现在请记住计数值可能会大于实际的文档数。

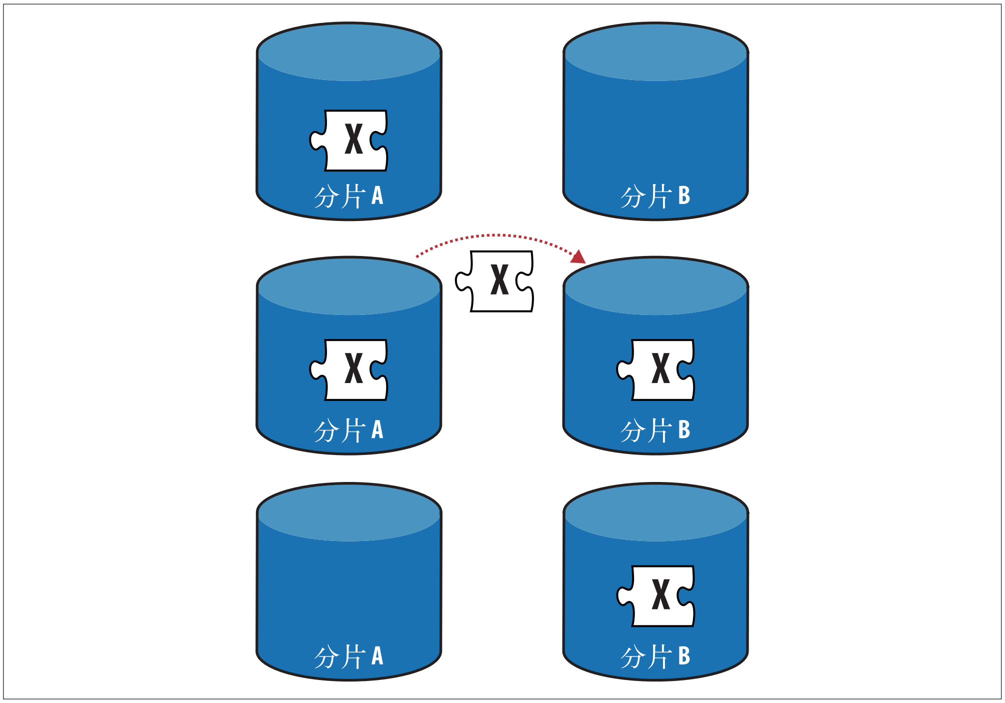


图 4-1：通过复制将一个块迁移到新的分片上，然后再从源分片上将它删除

4.2.2　唯一索引

假设我们在 email 字段上进行分片，同时还想在 username 字段上加一个唯一索引，这在集群中是不可能强制实施的。

比方说我们有两个应用程序服务器负责处理用户信息，其中一个添加了包含如下字段的新用户文档：

```
{
    "_id" : ObjectId("4d2a2e9f74de15b8306fe7d0"),
    "username" : "andrew",
    "email" : "awesome.guy@example.com"
}
```

要检查“andrew”是否为集群中唯一的“andrew”，需要遍历每一台服务器上每个文档的 username 字段。假设 MongoDB 遍历了所有分片发现没有其他人再叫“andrew”，正当它要把文档写入分片 3 时，第二个应用程序服务器发来了下面这个要被插入的文档：

```
{
    "_id" : ObjectId("4d2a2f7c56d1bb09196fe7d0"),
```

```
    "username" : "andrew",
    "email" : "cool.guy@example.com"
}
```

又一次，每个分片都要检查确认自己没有名为“andrew”的用户。当然还不会有，因为第一份文档还没有被写入，这样分片 1 继续并写入了这个文档。接着分片 3 终于有空完成第一份文档的写入。现在就出现两个用户名相同的人了！

通常情况下，要确保分片间不存在重复，唯一的方法是在每次要执行写操作时锁住整个集群，直至写操作确认成功。这样做系统很难实现高性能写。

因此，除片键以外任何键都难以保证唯一性。可以保证片键的唯一性是因为特定文档只能属于一个数据块，所以它只需要在那个分片上唯一，就能保证在整个集群中是唯一的。你也可以有一个以片键开头的唯一索引。举个例子，如果我们在用户集合上按 `username` 分片，和上面一样，但是包含唯一性选项，那么就可以在 `{username : 1,email : 1}` 上创建一个唯一索引。

由此推出一个有趣的结论：除非在 `_id` 上分片，否则能够创建出不唯一的 `_id` 值。尽管不推荐这么做（而且还会在移动块时带来麻烦），但却是可能的。

4.2.3 更新

更新操作默认只针对单个记录。这意味着它们和唯一索引面临同样的麻烦，即没有什么好办法能确保操作在多个分片间只发生一次。因此如果要更新单个文档，一定要在条件中使用片键（`update` 的第一个参数）。如果不这么做，你会得到一个错误。

```
> db.adminCommand({shardCollection : "test.x", key : {"y" : 1}})
{ "shardedCollection" : "test.x", "ok" : 1 }
>
> // 可以运行
> db.x.update({y : 1}, {$set : {z : 2}}, true)
>
> // 出错
> db.x.update({z : 2}, {$set : {w : 4}})
can't do non-multi update with query that doesn't have the shard key
```

批量更新中可以使用任何条件。

```
> db.x.update({z : 2}, {$set : {w : 4}}, false, true)
> // 没有错误
```

如果碰到了奇怪的错误信息，考虑一下执行的操作是否对整个集群必须是原子化的。这类操作是不被允许的。

4.3 MapReduce

在集群上运行 MapReduce 时，每个分片都会执行它自己的映射（map）及收敛（reduce）。*mongos* 选出一个“领头羊”分片以接受来自其他分片的收敛结果并在此完成最后的收敛。一旦数据被收敛到最终形态，就会按照指定的方式输出。

由于分片将任务分解到多台机器，因此能比单台服务器更快地执行 MapReduce。尽管如此，它仍不适合做实时运算。

临时集合

1.6 版里，除非声明了“out”参数，否则 MapReduce 会创建临时集合。这些临时集合会一直存在直至创建它们的连接关闭。在单台服务器上这种工作方式运行得很好，但是 *mongos* 会维护自己的连接池而且从不关闭对分片的连接。因此，临时集合永远都不会被清除掉（因为创建它们的连接从未被关闭），并且它们会一直存在，数量越来越多。

如果你正在运行 1.6 版本同时还使用了 MapReduce，就必须手动清理所有临时集合。在指定数据库中，你可以通过执行下面的函数来删除所有临时集合：

```
var dropTempCollections = function(dbName) {
    var target = db.getSisterDB(dbName);
    var names = target.getCollectionNames();

    for (var i = 0; i < names.length; i++) {
        if (names[i].match(/tmp\.mr\./)){
            target[names[i]].drop();
        }
    }
}
```

后续版本中，MapReduce 会强制要求你对输出进行处理。详情请查看文档。[1]

译注1：1.7.4及以后版本中，`out`不再是可选参数，其中各种可用的参数形式及其含义具体见http://www.mongodb. org/display/DOCS/MapReduce。

第 5 章

管理

上一章主要从应用程序开发者角度来介绍使用 MongoDB 的有关内容，而本章将更多地从运维方面介绍有关集群管理的内容。集群启动并运行起来之后，如何了解其运行情况呢？

5.1 使用命令行

对单个 MongoDB 实例来说，对集群的大部分管理工作都可以通过 *mongo* shell 来完成。

5.1.1 了解概况

`db.printShardingStatus()` 可以给出一份执行概况。它能收集所有与集群有关的重要信息并漂亮地展示出来。

```
> db.printShardingStatus()
--- Sharding Status ---
sharding version: { "_id" : 1, "version" : 3 }
shards:
{ "_id" : "shard0000", "host" : "ubuntu:27017" }
{ "_id" : "shard0001", "host" : "ubuntu:27018" }
databases:
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "test", "partitioned" : true, "primary" : "shard0000" }
test.foo chunks:
shard0001 15
shard0000 16
{ "_id" : { $minKey : 1 } } -->> { "_id" : 0 } on : shard1 { "t" : 2, "i" : 0 }
{ "_id" : 0 } -->> { "_id" : 15074 } on : shard1 { "t" : 3, "i" : 0 }
{ "_id" : 15074 } -->> { "_id" : 30282 } on : shard1 { "t" : 4, "i" : 0 }
{ "_id" : 30282 } -->> { "_id" : 44946 } on : shard1 { "t" : 5, "i" : 0 }
{ "_id" : 44946 } -->> { "_id" : 59467 } on : shard1 { "t" : 7, "i" : 0 }
{ "_id" : 59467 } -->> { "_id" : 73838 } on : shard1 { "t" : 8, "i" : 0 }
…… 这里省略了几行 ……
{ "_id" : 412949 } -->> { "_id" : 426349 } on : shard1 { "t" : 6, "i" : 4 }
{ "_id" : 426349 } -->> { "_id" : 457636 } on : shard1 { "t" : 7, "i" : 2 }
{ "_id" : 457636 } -->> { "_id" : 471683 } on : shard1 { "t" : 7, "i" : 4 }
{ "_id" : 471683 } -->> { "_id" : 486547 } on : shard1 { "t" : 7, "i" : 6 }
{ "_id" : 486547 } -->> { "_id" : { $maxKey : 1 } } on : shard1 { "t" : 7, "i" : 7 }
```

`db.printShardingStatus()` 可以打印出一个包含所有分片和数据库的列表。每个分片集合一个条目（这儿只有一个分片集合 *test.foo*）。它会显示出数据块的分布情况（`shard0001` 上 15 个块，而 `shard0000` 上 16 个块）。然后是每个块的详细信息，包括它的区间（比如 `{ "_id" : 115882 } -->> { "_id" : 130403 }` 对应 `_id` 在区间 [115882, 130403）里的文档）和它所在的分片。它还会给出块的主次版本号，这个不需要关心。

创建的每个数据库都有一个主分片作为“大本营”。在这个例子里，*test* 数据库的主分片被随机设定为 shard0000。这并不意味任何东西——shard0001 最后比 shard0000 的数据块还多！这个字段应该永远与你无关，所以可以忽略它。如果删除掉某个数据库的主分片，则其主分片会被自动地移动到集群中另一个分片上。

集合很大的话，`db.printShardingStatus()` 的输出就会特别长，因为要列出每个分片上的每个块。如果集群非常大，通过深入研究能够得到更多准确的信息，但是如果你刚上道，那么这个漂亮的概览正合适。

5.1.2　配置集合

mongos 会将请求分发到适当的分片上，除非查询的是 *config* 数据库。访问 *config* 数据库会直接转到配置服务器上去，而那正是能找到所有集群配置信息的地方。如果你的集合由几百或几千个数据块组成，那就值得了解一下 *config* 数据库的内容，以便能够查询特定信息，而非获得整个集群的概况。

让我们看看 *config* 数据库。假设你有一个集群，应该能看见以下集合：

```
> use config
switched to db config
> show collections
changelog
chunks
collections
databases
lockpings
locks
mongos
settings
shards
system.indexes
version
```

其中很多都对应集群 *config.mongos*（所有 *mongos* 进程的列表，包括过去和现在的）的组成部分。

```
> db.mongos.find()
{ "_id" : "ubuntu:10000", "ping" : ISODate("2011-01-08T10:11:23"), "up" : 0 }
{ "_id" : "ubuntu:10000", "ping" : ISODate("2011-01-08T10:11:23"), "up" : 20 }
{ "_id" : "ubuntu:10000", "ping" : ISODate("2011-01-08T10:11:23"), "up" : 1 }
```

`_id` 是 *mongos* 的主机名。`ping` 是配置服务器最后一次 ping 它的时间。`up` 是它认为 *mongos* 是否在线。如果启动了一个 *mongos*，即使只是几秒钟，它都会被添加到这个列表中且不会消失。这并不重要，因为你不太可能会启动数百万个 *mongos* 服

务器，不过还是需要提醒一下，这样看到列表时才不会感到困惑。

config.shards——集群中的所有分片。
config.databases——集群中的所有数据库（包括分片版和未分片版）。
config.collections——所有分片集合。
config.chunks——集群中的所有数据块。

config.settings 包含（理论上）与数据库版本相关的可调设置。目前，*config.settings* 允许调整块大小（但是别这么做！）和关闭平衡器，通常情况下都不需要这么做。可以通过执行一个更新来修改这些设置。比如，要关掉平衡器：

```
> db.settings.update({"_id" : "balancer"}, {"$set" : {"stopped" : true }}, true)
```

如果正处在一轮平衡化过程中，则直到完成为止平衡器都不会关闭。

剩下的集合中唯一可能需要关注的就是 *config.changelog*。它是一个非常详尽的日志，其中记录了每一次发生的分割和迁移，可以用于跟踪使集群变成当前配置状态的步骤。不过，通常它都比需要的更详尽。

5.1.3　应该连接什么

如果执行的只是正常的读写或管理操作，答案永远是“*mongos*”。它可以是任何 *mongos*（记住它们是无状态的），但永远是 *mongos*，而不是分片，也不是配置服务器。

如果操作并不寻常，可能就要连接配置服务器或分片了。也许是为了直接查看某个分片的数据或是手动修改一个被搞乱的配置。比如，你必须直接连到分片上才能修改副本集的配置。

记住配置服务器和分片都只是一般的 *mongod* 而已。你所知道的任何在 *mongod* 上执行的操作都同样适用于配置服务器和分片。尽管如此，在正常的运维过程中，应该几乎永远不需要连接它们。所有正常的操作都应该通过 *mongos* 完成。

5.2　监控

当你有一个集群时，监控非常重要。所有对监控单个节点的建议都适用于监控多个节点，因此请确保已读过有关监控的文档。

注意，当机器数量变多时，网络会变得越来越重要。如果一台服务器表示连不上另一台服务器，要检查两台服务器间网络的连通性。

如果可能，请保留一个已经连上集群的 shell。创建一个连接要求 MongoDB 暂时给

连接一个锁，这在调试时可能是个问题。假设一台服务器出状况了，因此你启动一个 shell 去连它。非常遗憾的是，*mongod* 正好卡在写锁上，因此 shell 会一直尝试获取锁，进而永远也不能完成连接。所以为了安全起见，留下一个打开的 shell。

5.2.1 mongostat

mongostat 是可用监控工具中最全面的。它能返回大量与服务器运行有关的信息，包括负载、页错误及打开的连接数。

如果有一个集群，你可以为每台服务器单独启动一个 *mongostat*，也可以在某个 *mongos* 上运行 `mongostat --discover` 并让它来分析出集群的所有成员并显示其状态。

举个例子，如果我们使用第 4 章描述的 *simple-setup.py* 脚本启动了一个集群，它能找出所有的 *mongos* 进程和分片：

```
$ mongostat --discover
                   mapped   vsize   res  faults locked%  idx miss%  conn   time       repl
localhost:27017        0m    105m    3m       0        0         0     2   22:59:50   RTR
localhost:30001       80m    175m    5m       0        0         0     3   22:59:50
localhost:30002        0m     95m    5m       0        0         0     3   22:59:50
localhost:30003        0m     95m    5m       0        0         0     3   22:59:50

localhost:27017        0m    105m    3m       0        0         0     2   22:59:51   RTR
localhost:30001       80m    175m    5m       0        0         0     3   22:59:51
localhost:30002        0m     95m    5m       0        0         0     3   22:59:51
localhost:30003        0m     95m    5m       0        0         0     3   22:59:51
```

我简化了输出并删除了一些列，因为每行只能放下 80 个字符，而 *mongostat* 一行有足足 166 个字符宽。另外空行看上去也有点奇怪，因为这个工具以"正常"*mongostat* 空行开始，列出集群的剩余部分，还要再添加两个字段 *qr|qw* 和 *ar|aw*。这些字段分别显示了有多少连接在读和写操作上排队等待以及有多少处于活动状态的读写操作。

5.2.2 Web管理界面

如果使用副本集，请确保使用 `--rest` 选项启动它们。副本集的 Web 管理界面（*http://localhost:28017/_replSet*，如果 *mongod* 是在端口 27017 上运行的）可以给出大量的相关信息。

5.3　备份

在运行的集群上做备份其实是非常有难度的。数据不停地被应用程序添加和删除，同时又照例被平衡器挪来挪去。若今天转储一个分片明天再恢复，就有可能有某些文档同时出现在两处或者彻底丢失（见图 5-1）的情况。

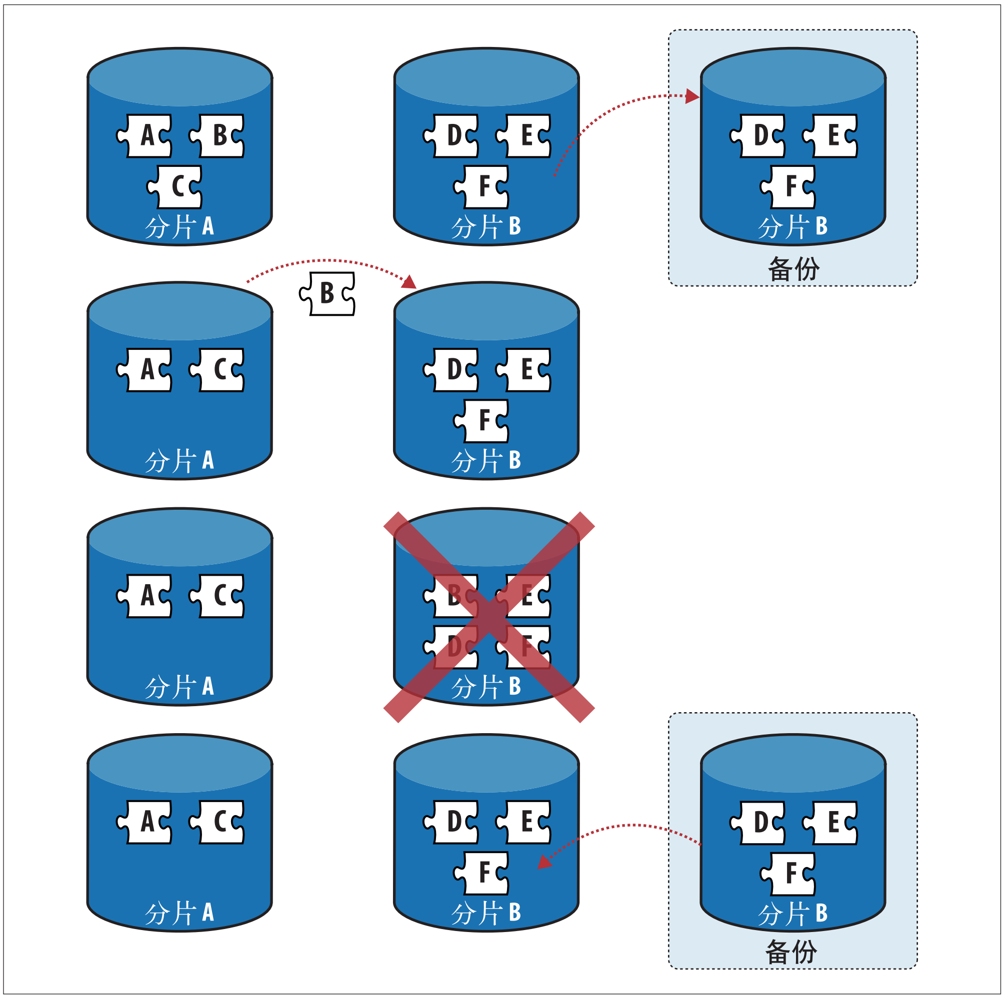


图 5-1：迁移前做了备份。如果迁移后分片崩溃并从备份中恢复，则集群会失去被迁移的块

创建备份的问题在于通常你只想要恢复集群的一部分（你不会想从昨天的备份中恢复整个集群，而只是想恢复停机的那个节点）。如果要从备份中恢复，必须小心谨慎。首先查看一下配置服务器，明确哪些数据块应该在要被恢复的分片上，然后只从备份中恢复这些块的数据（还有 *mongorestore*）。

如果想要整个集群的一个快照，那就必须关闭平衡器，*fsync* 并锁住集群中的所有从机，转储，然后对其解锁并重启平衡器。通常人们只是从个别分片上提取备份。

5.4 关于架构的建议

你可以创建一个分片集群然后不再动它，但常规维护怎么做？为了让集群更容易管理，可以在架构上多花些心思。

5.4.1 创建应急站点

名字暗示了正在运行的是一个 Web 站点，不过这一方法对多数类型的应用程序都适用。如果需要让应用偶尔下线（比如进行系统维护、更改，或者处于紧急情况时），有个可以切换过去的应急站点就显得非常便利了。

应急站点完全不应使用自己的数据库集群。如果要使用某个数据库，则此数据库应当与主数据库完全断开连接。你也可以让它通过缓存提供数据，要不就把整个应急站点做成静态的，这取决于应用程序。不管怎样做一点东西给用户看总比显示 Apache 的错误页面好。

5.4.2 挖护城河

有一个好办法能阻止或最小化各类问题，那就是在集群服务器周围挖出一条“虚拟护城河”，然后通过队列控制对集群的访问。

队列可以让应用程序在计划的中断运行时继续处理写入，或者至少避免在中断运行开始前丢失那些尚未完成的写入。你可以将它们保存在队列中直到 MongoDB 再次启动为止，然后再把它们发送给 *mongos*。

队列不仅在容灾中非常有用，而且在常规的突发流量下也非常有用。队列可以吸收短时间内爆发的大量请求并释放出一个和缓稳定的请求流，而不是让突发的请求洪流淹没集群。你也可以把队列反过来用，即缓存 MongoDB 返回的结果。

有很多不同的队列可供使用，包括 Amazon SQS、RabbitMQ，甚至是一个 MongoDB 集合（当然一定得把它和被保护的集群分开来），随便哪种只要用起来舒坦就行。

不过，队列也不是对所有类型的应用都有效，比如对需要实时数据的应用就没什么帮助。尽管如此，如果应用程序可以忍受微小的延迟，队列能够在外部世界和数据库之间架起一座非常有价值的桥梁。

5.5 错误处理

正如第 1 章提到的，网络分区（network partition）、服务器崩溃以及其他问题能够引发各式各样的麻烦。很多情况下 MongoDB 可以（至少暂时）从这类麻烦中“自愈”。这一节会介绍遇到哪些问题可以不用处理尽管放心大睡，而对哪些问题却不能置之不理，同时介绍如何让应用程序准备好应对各种运行中断。

5.5.1 分片停机

如果一整个分片都停机了，则所有应该命中该分片的读写都会返回错误。应用程序应该处理这种错误（无论使用何种编程语言，这都如同在遍历游标时抛出的异常）。举例来说，如果一个查询的头 3 个结果在运行正常的分片上，而包含有用数据块的第四个分片却停机了，你会得到这样的输出：

```
> db.foo.find()
{ "_id" : 1 }
{ "_id" : 2 }
{ "_id" : 3 }
error: mongos connectionpool:
connect failed ny-01:10000 : couldn’t connect to server ny-01:10000
```

对这种错误要有所准备，以便出现问题时尽可能让应用平滑地运行下去。就有些程序而言，也可以有针对性地继续查询直至分片重新上线为止。

未来 MongoDB 会增加对部分查询结果的支持（post-1.8.0），这样就可以只从正常运行中的分片返回结果而不指出发生的任何异常。

5.5.2 多数分片停机

如果用副本集做分片，则理想情况下整个分片不会全停掉，而仅仅是其中的一两台服务器。如果副本集失去了多数成员，则没有服务器能成为主机（除非重新进行人工调整），集合就会变成只读状态。[1] 一旦它变为只读状态，就要确保应用程序仅发送给它带有 `slaveOkay` 选项的读请求。

如果使用了副本集，则理想情况下单个服务器（甚至是一些服务器）出错完全不会影响到应用程序。集合中剩余的服务器会承担起负载，而应用甚至不会注意到发生的变化。

> 在 1.6 版中，如果一个副本集配置发生变化，日志里会出现大量重复消息。因为 *mongos* 和分片间的每一条连接都在注意到副本集连接已过期时执行更新，而它更新时就会打印一条消息。尽管如此，这应该不会对运行造成影响，而只是有些啰嗦吵闹而已。在 1.8 版本中这个问题已经被修复了，那之后在更新副本集配置的事情上，*mongos* 变得聪明多了。

译注1：2.0版本已发生变化，详情见http://www.mongodb.org/display/DOCS/2.0+Release+Notes#2.0Release-NotesReconfigurationwithaminorityup。

5.5.3　配置服务器停机

一台配置服务器停机并不会立即对集群的性能产生影响，但是却会导致无法修改任何配置。所有配置服务器必须协同工作，因此只要是一个兄弟倒下了，剩下的配置服务器都不再允许任何修改。要注意的是，配置服务器停机时不能修改任何配置，比如不能添加 mongos 服务器，也不能迁移数据，也不能添删数据库或集合，当然也不能修改副本集的配置。

如果一台配置服务器崩溃了，务必要让它重启以便能在需要的时候修改配置，但它应该完全不会影响到集群当时的操作。请确保监控了配置服务器，并且如果一台坏掉了，就让它尽快恢复。

数据迁移期间配置服务器坏掉是会给服务器带来一些压力的。迁移的最后几步之一就是更新配置服务器。因为只要一个服务器停机了就全部无法更新，所以分片不得不回退迁移并删除辛辛苦苦复制完的所有数据。如果分片没有过载，那么不至于太悲剧，可惜有点浪费。

5.5.4　mongos进程死掉

由于你总是可以有很多 *mongos* 进程，而且它们没有状态，所以如果其中一个坏掉了也不是什么大事。推荐在每个应用程序服务器上跑一个 *mongos* 并且让每个应用程序服务器和它本地的 *mongos* 通信（见图 5-2）。这样的话，如果整个机器停掉，也不会有人尝试与一个不存在的 *mongos* 进程对话。

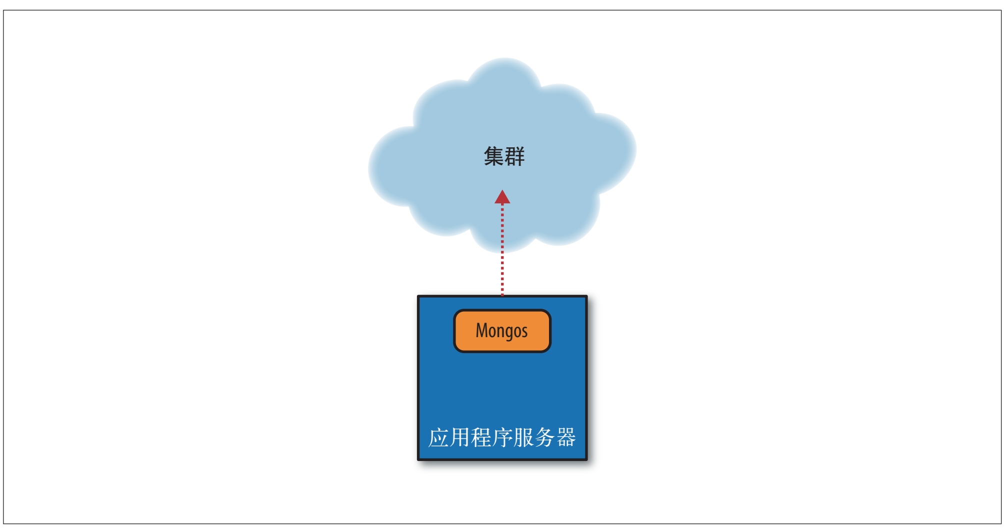


图 5-2：运行了 mongos 的应用程序服务器

预备一些冗余的 *mongos* 服务器，这样如果一个 *mongos* 进程坏掉而应用程序服务器还正常，就可以将故障转移到其他 *mongos* 上。大部分驱动允许声明一组可连接的服务器并会按顺序尝试连接。因此你可以把首选的 *mongos* 放在前面，后面跟着备用的 *mongos*。如果一个坏掉了，应用程序可以处理异常（用你使用的任何语言），然后驱动会自动为下一个请求切换后备服务器。

由于 *mongos* 无状态且不保存数据，如果机器是好的，你也可以只是尝试重启崩溃掉的 *mongos*。

5.5.5 其他注意事项

以上各点在处理时都应该与其他任何可能出问题的地方隔离开来。有时，由于网络分区你有可能会失去全部分片，或一部分其他分片或配置服务器或者 *mongos* 进程。建议你从面向用户（用户还能做什么？）和程序设计（应用程序还能切合实际地做什么？）两个角度审慎考虑如何应对各种场景。

最后，MongoDB 在显露功能丧失症状前会尽力承受各种错误。当然，如果遇上了完美风暴（一定会），难免会失去功能，但是平日里的服务器崩溃，电力短缺以及网络分区不应该造成太大的问题。敬请死盯监控保持镇定，呵护好你的集群。

第 6 章

学习资源

如果你按照前面各章给出的意见做了，那么就应该能够顺利地构建一个高效且可预测的分布式系统，还能按需增长。如果你还有问题和其他困惑，请发电子邮件给我，我的邮箱是 kristina@10gen.com。

如果你有兴趣进一步学习分片，这里有些可用的资源。

- MongoDB 的 wiki 有很大篇幅介绍了分片的内容，包括了从配置示例到内部机制的详细内容。
- MongoDB 用户列表是一个提问的好地方。
- 在 mongo-snippets 源码库里有很多有用的代码片段。
- Boxed Ice 在生产环境中部署了一个 MongoDB 集群并且经常在博客上写一些与运行 MongoDB 相关的实用文章。
- 如果你有兴趣阅读更多有关分布式计算的理论，我强烈推荐 Leslie Lamport 关于 Paxos 算法论文，该论文非常有趣且有启发性。

另外，如果你喜欢这本书，建议你阅读我的博客，其中主要是一些关于 MongoDB 的高级话题。

MongoDB开发技巧50例

50 Tips and Tricks for MongoDB Developers



［美］Kristina Chodorow 著
程显峰 译

前言

MongoDB 上手很容易，但是一旦用它来构建应用，一些棘手的问题便会接踵而来。究竟该怎样设计数据架构？拆分成两个文档还是放在一块儿？性能如何提高？本书正是用来解答这些问题的。

本书涵盖的技巧分为若干主题：
第 1 章讲述数据库设计的方方面面；
第 2 章讲的是编写应用程序时应该注意的问题；
第 3 章旨在为应用提速；
第 4 章讲述怎样做到在不太损失性能的前题下提高数据的安全性，主要利用复制和日志来做到这一点；
第 5 章是配置和运维的经验总结。

很多技巧涉及多章内容，尤其是与性能相关的。第 3 章主要针对索引进行讲述，但是性能涉及诸多方面，不论是数据设计、实现，还是数据安全都会对其有影响。

读者对象

本书适合已经有些 MongoDB 基础知识的使用者。若是对 MongoDB 不了解，可以看看《MongoDB 权威指南》或者在线文档（http://www.mongodb.org）。

格式约定

本书使用了如下排版约定。

- 楷体

用于标记新名词。

• 等宽字体

用于程序代码，在段落中用于表示程序的组成部分，如变量或函数名、数据库、数据类型、环境变量、语句、关键字。

• 等宽粗体

命令或是其他应该由用户输入的内容。

• 等宽斜体

应该由用户提供的或根据上下文确定的值。

这个图标表示提示、建议或一般性的注解。

这个图标表示一个警告或警示。

使用示例代码

本书用于帮助你完成工作。通常，你可以在程序或文档中使用本书提供的代码。除非你在重新发布我们的大量代码，否则不需要联系我们来获得许可。比如，在程序中使用本书代码的一些片段是无需我们许可的。但是出售或再分发 O'Reilly 的图书示例光盘显然是需要授权的。引用本书或引用示例代码来回答问题是不需要授权的，但将本书的大量示例代码纳入产品的文档是需要授权的。

我们乐于见到你在使用时声明引用信息，但不强制要求。引用信息通常包括书名、作者、出版社和 ISBN，例如"*50 Tips and Tricks for MongoDB Developers* by Kristina Chodorow (O'Reilly). Copyright 2011 Kristina Chodorow, 978-1-449-30461-4"。

如果你认为对示例代码的使用需要授权，请通过这个邮箱联系我们：permissions@oreilly.com。

Safari®在线图书

Safari 在线图书是应需而变的数字图书馆。它能够让你非常轻松地搜索 7500 多种技术性和创新性参考书以及视频，以便快速地找到需要的答案。

订阅后你就可以访问在线图书馆内的所有页面和视频，在手机或其他移动设备上阅读，在新书上市之前抢先阅读，还能够看到尚在创作中的书稿并向作者反馈意见。复制粘贴代码示例、放入收藏夹、下载部分章节、标记关键点、做笔记甚至打印页面等有用的功能可以帮你节省大量时间。

这本书也在其中。欲访问本书的英文版电子版，或者由 O'Reilly 或其他出版社出版的相关图书，请到 http://my.safaribooksonline.com 免费注册。

我们的联系方式

请把对本书的评论和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreilly.com/catalog/9781449304614

中文书：

http://www.oreilly.com.cn/index.php?func=book&isbn=9787115272119

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

关于本书的更多信息、会议、资源中心和网络，请访问以下网站：

http://www.oreilly.com
http://www.oreilly.com.cn

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

第 1 章

应用设计技巧

1.1 技巧1：速度和完整性的折中

在多个文档中使用的数据既可以采用内嵌（反范式化）的方式，也可以采用引用（范式化）的方式。两种策略并没有优劣之分，各自都有优缺点，关键是要选择适合自己的应用场景的方案。

反范式化会产生不一致的数据。举例说明，假设要将图 1-1 中的苹果改成梨。假如仅仅更新了一个文档的值，应用就崩溃了，而你还没来得及更新其他文档，这时数据库中 `fruit` 就有两个不同的值。

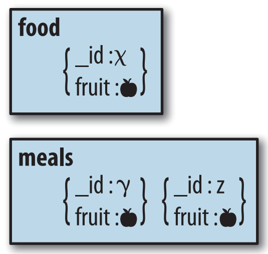


图 1-1：反范式化的设计。**`fruit`** 的值同时存储在 **`food`** 和 **`meals`** 集合里

不一致可不太好，但不太好的程度取决于到底存了什么。对于很多应用来说，短时的不一致还说得过去。如果有人更改了用户名，而他原有的博客文章在几个小时内还显示他的旧用户名也没什么大不了的。要是不能容忍哪怕一丁点不一致，则应该选用范式化的设计。

但要是做了范式化，应用则必须在每次确认水果时做额外一次查找（图 1-2）。若是应用无法承担这样的负载，又不太强调一致性，则应该反范式化。

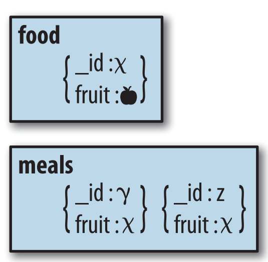


图 1-2：范式化的设计。**`fruit`** 字段存储在 **`food`** 集合里，被 **`meals`** 集合中的文档引用

这里需要的是两相权衡，因为极高的性能和瞬间一致性不可兼得。所以必须要想清楚哪个才是应用最需要的特性。

1.1.1 示例：网上购物车

假设现在要为购物车设计一下数据结构。MongoDB 中要存放应用的订单，但订单中该涵盖哪些信息呢？

范式化的设计

商品：

```
{
    "_id" : productId,
    "name" : name,
    "price" : price,
    "desc" : description
}
```

订单：

```
{
    "_id" : orderId,
    "user" : userInfo,
    "items" : [
        productId1,
        productId2,
        productId3
    ]
}
```

每个订单文档中存放商品的 `_id`。这样，当需要显示订单内容时，可以查询订单集合获得相应订单，之后查询商品集合获得指定 `_id` 的商品信息。这种设计下，一次查询无法获取完整的订单。

若有商品信息更新了，所有引用此商品的文档都会“更新”，因为这些文档只是指向了保存商品信息的文档。

范式化的结果就是读取速度较慢，但所有订单的一致性会有保证。多个文档会原子性地更新（因为仅仅引用的文档会更新）。

反范式化设计

商品（与前面相同）：

```
{
    "id" : productId,
```

```
        "name" : name,
        "price" : price,
        "desc" : description
    }
```

订单：

```
    {
        "_id" : orderId,
        "user" : userInfo,
        "items" : [
            {
                "_id" : productId1,
                "name" : name1,
                "price" : price1
            },
            {
                "_id" : productId2,
                "name" : name2,
                "price" : price2
            },
            {
                "_id" : productId3,
                "name" : name3,
                "price" : price3
            }
        ]
    }
```

这里将商品信息作为内嵌文档存在订单数据中。这样，当显示订单的时候只需一次查询。

如果商品信息变化了，需要更新，就要改动所有相应的订单。

反范式化读取速度较快，一致性稍弱。商品信息的变更不能原子性地更新到多个文档。

综上所述，你决定选哪个了吗？

1.1.2 考虑因素

主要应考虑以下几点。

- *是否总是要额外读取一次几乎不怎么改变的数据？*可能读了商品信息一万次才会修改一次它的详细信息。为了那一次写入快一点或者保证一致性，搭上一万次的读取消耗值吗？绝大多数应用都具有高读写比，不妨测算一下应用的读写比。

你认为引用的数据多久会更新一次？更新越少，越适合反范式化。有些极少变化的数据几乎根本不值得引用，例如名字、生日、股票代码、地址。

- 一致性很重要么？如果是肯定的，则应该范式化。例如，多个文档可能需要原子性地更新。如果设计的是一个交易系统，一些特定的证券必须在指定的时间交易，我们就会希望在它们不能交易时瞬时将其全部“锁”起来。为此就可以用一个锁文档来引用这组证券文档。但这种操作最好在应用层处理，因为应用需要知道何时上锁何时解锁。

  另外，当应用中的不一致性难以被容忍时也必须首先考虑一致性。在订单的例子中，层次比较严格：订单从商品中获取信息，但商品不会从订单中获取信息。假如有多个“信息源”文档，就比较难取舍了。

  然而，这个（构造出来的）订单系统中，一致性反而有不利的一面。假设我们希望将某个商品打八折。此时我们不想改变已有的订单的任何信息，仅仅是想更新一下商品的描述。这里我们要的是某一时刻的数据快照（参见技巧 5）。

- 要不要快速的读取？若想读取尽可能快，则要反范式化。在这个应用中这就无所谓了，所以不能算是考量因素。实时的应用要尽可能地反范式化。

订单文档非常适合反范式化，因为其中的商品信息不经常变化，就算变了也不必更新到所有订单。范式化在此就没什么优势可言了。

所以，本例中最佳选择是将订单反范式化。

延伸阅读：

- “Your Coffee Shop Doesn’t Use Two-Phase Commit”（咖啡店不需要两步提交，参见 http://www.eaipatterns.com/docs/IEEE_Software_Design_2PC.pdf）举例说明了现实世界中的系统怎样处理一致性，及其与数据库设计的关系。

1.2 技巧2：适应未来的数据要范式化

范式化可使数据可用性更长久，将来可以在不同的应用中以不同的方式查询范式化的数据。

这里的前提是有些数据将会年复一年不断地被各种应用用到。的确有这种数据，但是大多数人的数据都会不断地演化，陈旧的数据要么被更新，要么就被丢弃。绝大多数人都希望数据库能尽可能地对当前查询做出快速响应，如果将来查询变了，他们会针对新的查询优化数据库。

还有，应用如果很成功，其数据经常会高度定制化。当然，这并不是说这些数据不能用在别处，而且你很可能至少想要对数据做些元分析。但这和“适应未来”不同，“适应未来”代表满足未来 10 年内的所有查询。[1]

1.3　技巧3：尽量单个查询获取数据

> 本节会涉及一个术语——*应用单元*。对于 Web 应用或者移动应用，可以将对后端的一次请求视作一个应用单元。类似的例子还有：
>
> - 对于桌面应用，一次用户交互算作一个应用单元；
> - 对于分析系统，一个图表的加载算作一个应用单元。
>
> 应用程序中这些离散的应用单元涉及与数据库的交互。

MongoDB 的数据库设计要从应用单元的查询出发。

1.3.1　示例：博客

若要设计博客系统，对一篇博客文章的请求或许就是一个应用单元。显示一篇文章时，需要显示内容、标签、作者信息（虽然可能不是全部个人信息）以及评论。所以，这里将这些信息都嵌到文章文档中，以便通过一次查询就能获得所有必要信息。

记住是每页一次查询，不是一个文档一次查询，有时需要返回多个文档，或者文档的一部分（并非文档的所有内容）。例如，主页上要显示 `posts` 集合中最近的 10 篇文章，但只显示它们的标题、作者和概要：

```
> db.posts.find({}, {"title" : 1, "author" : 1, "slug" : 1, "_id" : 0}).sort(
... {"date" : -1}).limit(10)
```

可以把含有指定标签的最新 20 篇文章输出到一个页面：

```
> db.posts.find({"tag" : someTag}, {"title" : 1, "author" : 1,
... "slug" : 1, "_id" : 0}).sort({"date" : -1}).limit(20)
```

还有一个独立的 `authors` 集合，存放每位作者的详细信息。作者页面非常简单，仅需要从 `authors` 集合中读取一个文档：

```
> db.authors.findOne({"name" : authorName})
```

`posts` 集合中文档的一部分信息可能是作者文档中信息的一个子集（也许是作者名和头像）。

译注1：这就是Knuth所说的过早优化。

注意应用单元不一定非要对应单个文档，上面的一些例子仅仅算是巧合（一篇文章和一个作者页面都在单个文档中）。但是，还有很多情况下一个应用单元要对应多个文档，但只通过单条查询来完成这种请求。

1.3.2 示例：相册

假设有一个相册，用户可以创建或者回复含有照片和文字的帖子。有个应用单元是查看线索中的 20 条回帖，每个人的回复都是 posts 集合中一个独立的文档。要显示这个页面时，只需这样查询：

```
> db.posts.find({"threadId" : id}).sort({"date" : 1}).limit(20)
```

这样需要查询下一页的内容时，就查询紧接着的 20 条内容：

```
> db.posts.find({"threadId" : id, "date" : {"$gt" : latestDateSeen}}).sort(
... {"date" : 1}).limit(20)
```

还可以创建索引 {threadId : 1, date : 1} 来优化这种查询的性能。用 skip(20) 分页远不及范围高效，所以不予采纳。参见 *http://www.mongodb.org/display/DOCS/Advanced+Queries#Advanced Queries-{{skip%28%29}}*。

随着应用日渐复杂，用户和管理人员不断要求新功能，即使一个应用单元必须得多次查询，也不必担心。“一次查询”的目标是个良好的起点，可以很好地衡量初始的设计，但现实并非完美世界。稍微复杂的应用中，经常会有稀奇古怪的功能需要多次查询。

1.4 技巧4：嵌入关联数据

当在嵌入和引用文档之间犹豫不决时，不妨想想查询的目的究竟是为了获得字段本身的信息，还是为了进一步获取更广泛的信息。例如按照某个标签查询应该返回带有该标签的文章，而不仅仅是标签本身。类似地，对于查询评论，可能已存在一个最新评论列表，但人们更希望知道是哪篇文章引发了评论，除非你的应用就是以评论为核心的。

若是从关系型数据库迁移到 MongoDB，联结表就是重点要考虑嵌入的对象。那些仅有一个键和一个值的表（如标签、权限、地址）在 MongoDB 中几乎都应该做嵌入处理。

还有，若是某些信息只在一个文档中使用，则应该嵌进这个文档。

1.5 技巧5：嵌入时间点数据

技巧 1 中订单的例子已经提到，当一个商品打折或者换了图片，并不需要更改原来订单中的信息。类似这种特定于某一时刻的时间点数据，都应做嵌入处理。

订单文档中还有一处也是这样，地址也是这种时间点数据。若某人更新了个人信息，那么并不需要更改其以往的订单内容。

1.6 技巧6：不要嵌入不断增加的数据

MongoDB 存储数据的机制决定了对数组不断追加数据是很低效的。在正常使用中数组和对象大小应该相对固定。

所以，嵌入 20 个子文档，或者 100 个，或者 1 000 000 个都不是问题，关键是提前这么做，之后基本保持不变。放任文档增长会使系统慢得让你受不了的。

评论是个比较特殊的地方，因应用不同而差别较大。对大多数应用而言，评论应该内嵌到所依附的父文档中。但是，有些系统中评论本身自成条目，或者动辄成百上千，这种情况就应该将其放到独立的文档中了。

还有个例子，假设正在做一个以评论为核心的应用。技巧 3 中相册的例子与此有点类似，主要的内容就是评论，这时将其作为单独的文档处理比较合适。

1.7 技巧7：预填充数据

如果已经知道未来要用到哪些字段，第一次插入时就带着这些字段会比用到时再创建效率更高。例如，做一个网站分析的应用，以查看一天当中每一分钟有多少用户查看不同的页面。设置一个 `pages` 集合，其中每个文档表示一个页面 6 个小时的信息。而且要按分钟和小时两种频率来存储信息：

```
{
    "_id" : pageId,
    "start" : time,
    "visits" : {
        "minutes" : [
            [num0, num1, ..., num59],
            [num0, num1, ..., num59],
            [num0, num1, ..., num59],
            [num0, num1, ..., num59],
            [num0, num1, ..., num59],
            [num0, num1, ..., num59]
        ],
        "hours" : [num0, ..., num5]
    }
}
```

这样做有个明显的好处——可以始终知道文档的形式。这个文档有一个以当前时间为开始时间的字段，未来 6 小时每一分钟的数据都会记录在另一个字段中，然后是一个接一个的类似文档。

因此，可以在系统负载较低的时段批量插入模板文档，或者将这个工作均匀地散布到一天当中。插入的文档具有类似下面的结构，将 someTime 替换为接下来 6 小时的开始时间：

```
{
    "_id" : pageId,
    "start" : someTime,
    "visits" : {
        "minutes" : [
            [0, 0, ..., 0],
            [0, 0, ..., 0],
            [0, 0, ..., 0],
            [0, 0, ..., 0],
            [0, 0, ..., 0],
            [0, 0, ..., 0]
        ],
        "hours" : [0, 0, 0, 0, 0, 0]
    }
}
```

这样，当增加或者设置这些计数器的时候，MongoDB 就不用为其分配空间了，只更新已输入的值，这要快得多。

例如，在某一小时开始的时候，可以这样更新：

```
> db.pages.update({"_id" : pageId, "start" : thisHour},
... {"$inc" : {"visits.0.0" : 3}})
```

这一想法也适用于其他类型的数据，甚至集合和数据库也可以用类似的方法。若每天都要使用新的集合，最好也提前创建。

1.8　技巧8：尽可能预先分配空间

这个技巧与技巧 6 和技巧 7 关系紧密。只要知道文档开始比较小，后来会变为确定的大小，就可以使用这种优化方法。一开始插入文档的时候，就用和最终数据大小一样的垃圾数据填充，即添加一个 garbage 字段（其中包含一个字符串，串大小与文档最终大小相同），然后马上重置字段。

```
> collection.insert({"_id" : 123, /* other fields */, "garbage" : someLongString})
> collection.update({"_id" : 123}, {"$unset" : {"garbage" : 1}})
```

这样，MongoDB 就会为文档今后的增长分配足够的空间。（参见图 1-3。）

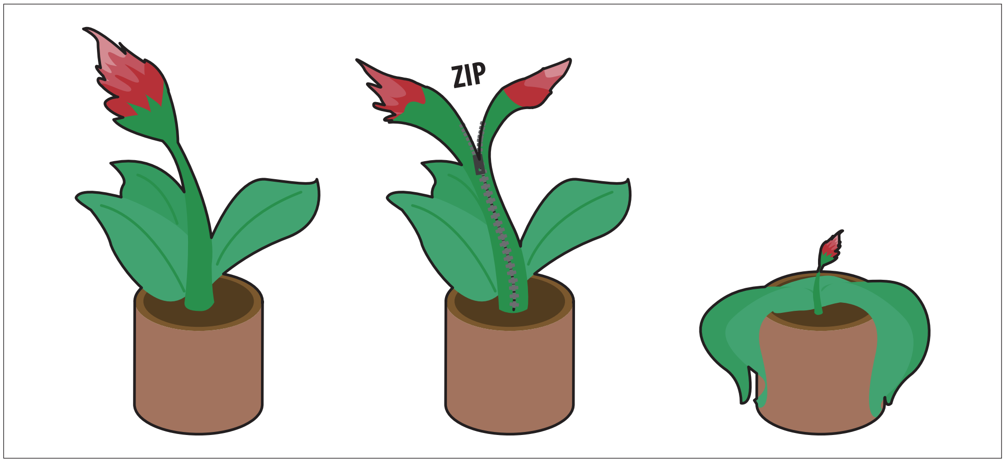


图 1-3：如果存储文档时便为其分配将来所需的空间，之后便不再需要移动它

1.9　技巧9：用数组存放要匿名访问的内嵌数据

一个常见的问题就是内嵌的信息到底是用数组存还是用子文档存。如果确切知道要查询的内容，则要用子文档。如果有时不太清楚查询的具体内容，则要用数组。当知道一些条目的查询条件时，通常该使用数组。

假设要编写一款游戏，其中的玩家可以挑选各种物品。我们可以对玩家的文档这样建模：

```
{
    "_id" : "fred",
    "items" : {
        "slingshot" : {
            "type" : "weapon",
            "damage" : 23,
            "ranged" : true
        },
        "jar" : {
            "type" : "container",
            "contains" : "fairy"
        },
        "sword" : {
            "type" : "weapon",
            "damage" : 50,
            "ranged" : false
        }
    }
}
```

假设要找出所有伤害值（damage）大于 20 的武器。子文档不支持这种查找物品

（items）的方式：“给我所有伤害值大于 20 的物品”。你只能知晓具体某种物品的信息，如“items.slingshot.damage（弹弓的伤害值）大于 20 吗”或“items.sword.damage（剑的伤害值）大于 20 吗”等。

如果想无需标识符就能获得某项的信息，就要用数组：

```
{
    "_id" : "fred",
    "items" : [
        {
            "id" : "slingshot",
            "type" : "weapon",
            "damage" : 23,
            "ranged" : true
        },
        {
            "id" : "jar",
            "type" : "container",
            "contains" : "fairy"
        },
        {
            "id" : "sword",
            "type" : "weapon",
            "damage" : 50,
            "ranged" : false
        }
    ]
}
```

这样用一条查询 {"items.damage" : {"$gt" : 20}} 就可以了。如果需要多条件查询（比如伤害值和攻击范围），可以用 $elemMatch（见 http://www.mongodb.org/display/DOCS/Advanced+Queries#Advanced Oueries-%24elem Match）。

那么什么时候该用子文档呢？字段名总是已知的时候最适合了。

例如，假设我们要了解玩家的属性，包括力量（str）、智力（int）、经验（wis）、敏捷（dex）、健康（con）和魅力（cha）。此时，我们总是明确地知道要查询哪个属性，所以可以这样设计：

```
{
    "_id" : "fred",
    "race" : "gnome",
    "class" : "illusionist",
    "abilities" : {
        "str" : 20,
        "int" : 12,
        "wis" : 18,
        "dex" : 24,
        "con" : 23,
```

```
        "cha" : 22
    }
}
```

要查询某个属性时，可以用 abilities.str，和 abilities.con 之类的查询。我们绝不会查找哪个属性大于 20，总是会知道要查哪个属性。

1.10 技巧10：文档要自给自足

MongoDB 是一种“无脑”的大型数据存储。也就是说，MongoDB 几乎不做任何数据处理，仅仅存取数据。要尽量遵守这点，避免让 MongoDB 做些能在客户端完成的计算。即便是些“小”任务，像求平均值或求和，也要放在客户端去做。

如果要找的信息必须经过计算，且无法直接从文档中获得，有两种方式：

- 付出高昂的性能代价（强制 MongoDB 使用 JavaScript 计算，参见技巧 11）；
- 优化文档结构，使得这些信息能够从文档中直接获得。

通常，你只需通过文档直截了当地给出这些信息。

现在假设要找到苹果（apples）和橘子（oranges）总数为 30 的文档。文档结构如下：

```
{
    "_id" : 123,
    "apples" : 10,
    "oranges" : 5
}
```

像上面这样设计的话，查询总数的任务就得仰仗 JavaScript 了，结果会非常低效。但是可以在文档中加入一个总数（total）字段：

```
{
    "_id" : 123,
    "apples" : 10,
    "oranges" : 5,
    "total" : 15
}
```

apples 和 oranges 字段变化时总数也要变化：

```
> db.food.update(criteria,
... { “$inc” : { “apples” : 10, “oranges” : -2, “total” : 8}})
> db.food.findOne()
{
    "_id" : 123,
    "apples" : 20,
```

```
    "oranges" : 3,
    "total" : 23
}
```

如果不太确定更新会改变什么内容，情况就会变得复杂。例如，要查询水果的种类，但是不知道更新会不会增加新品种。

假设文档如下：

```
{
    "_id" : 123,
    "apples" : 20,
    "oranges : 3,
    "total" : 2
}
```

如果一个更新可能创建新的字段，也可能不创建，品种总数该不该增加呢？若是创建了新的字段，总数也是要随着增加的：

```
> db.food.update({"_id" : 123}, {"$inc" : {"banana" : 3, "total" : 1}})
```

相反，如果 banana 字段已经有了，则不应增加总数。但是客户端根本无从知晓其存在与否。

你恐怕已经猜到了，有两种解决办法，一种快但不保证一致，另一种慢但能保证一致性。

快速的做法就是简单选择给 total 加 1 或者不加，然后让应用在客户端验证实际的总数。可以做一个一直在后台运行的批处理任务来纠正这些不一致的地方。

如果应用时间不紧，可以用 findAndModify 来“锁”住文档（设定“locked”字段，其他写入会手动检查这个字段），返回文档，然后同时对文档解锁并更新相应的字段和 total：

```
> var result = db.runCommand({"findAndModify" : "food",
... "query" : {/* 其他条件 */, "locked" : false},
... "update" : {"$set" : {"locked" : true}}})
>
> if ("banana" in result.value) {
... db.fruit.update(criteria, {"$set" : {"locked" : false},
...     "$inc" : {"banana" : 3}})
... } else {
... // 如果 banana 不存在，则将 total 值递增
... db.fruit.update(criteria, {"$set" : {"locked" : false},
...     "$inc" : {"banana" : 3, "total" : 1}})
... }
```

究竟是用哪种得看具体情况。

1.11 技巧11：优先使用$操作符

$ 操作符应付不来某些操作。但是对于绝大多数应用，让文档本身提供足够的内容可以极大地简化查询的复杂度。然而，还是有些情况下的查询不能用 $ 操作符表达。这时候就得借助 JavaScript 了，可以在查询中使用 $where 子句来执行 JavaScript 代码。

查询中的 $where 对应一个 JavaScript 函数，该函数的返回值为 true 或者 false(表示匹配与否)。要是想查找所有 member[0].age 和 member[1].age 相等的文档，可以这样操作：

```
> db.members.find({"$where" : function() {
... return this.member[0].age == this.member[1].age;
... }})
```

正如预期，$where 为查询带来了极大的灵活性。但是也带来很大的效率问题。

1.11.1 深入了解

$where 效率不高是因为 MongoDB 为此要做很多事情。对于普通的查询（没有 $where 的查询），客户端将查询转换成 BSON（http：//www.bsonspec.org），然后发送给数据库。MongoDB 中的数据也是 BSON 格式，所以直接比较就可以了。这种查询非常高效。

若是非得用 $where 不可，MongoDB 就得将集合中的每一个文档都转换成 JavaScript 对象，解析文档的 BSON 并添加它们的所有字段到 JavaScript 对象中。然后执行 JavaScript，之后就销毁。所以极其消耗时间和资源。

1.11.2 提高性能

$where 在必要时很有用，但能避免应尽量避免。实际上，如果查询中有很多 $where，说明应该重新考虑数据库设计是否合理了。

若是 $where 确有必要，可以减少 $where 要匹配文档的数量来缓解性能损失。设置一些不用 $where 就能执行的查询条件，并将其放在最前面。$where 遍历的文档越少，需要的时间就越少。

还是使用上面的例子，既然要查看会员的年龄，那么必须得是多人会员（joint），比如家庭会员（family）：

```
> db.members.find({'type' : {$in : ['joint', 'family']},
```

```
... "$where" : function() {
...     return this.member[0].age == this.member[1].age;
... }})
```

这样单人会员就不在 $where 遍历范围内了。

1.12 技巧12：随时聚合

要尽可能用 $inc 随时计算聚合。例如技巧 7 中，分析程序要有按分钟和按小时的统计数据。增加按分钟统计的数据时，同时可以增加按小时统计的数据。

如果聚合需要进一步的计算（比如，计算每小时的平均查询次数），就要将数据存放到分钟字段中，然后由后台的批处理按分钟字段中的最新数据计算平均值。由于所有需要聚合的信息都在一个文档中，对较新的（未聚合的）文档，甚至可以把计算放到客户端。至于旧文档，应该都通过批处理计算完了。

1.13 技巧13：编写代码处理数据完整性问题

由于 MongoDB 无模式的特点和反范式化的优势，你需要注意应用数据的一致性问题。

很多 ODM [1] 有多种方法来确保不同级别的一致性。然而，还是会有一致性问题，如由于系统崩溃导致的数据不一致（技巧 1），由于 MongoDB 更新的限制导致的不一致（技巧 10）。要处理这些不一致，要有个脚本做数据验证。

按照本章的建议，可能要有几个 cron 任务，这取决于具体的应用，如下所示。

一致性修复器

核对计算，检查重复数据，确保数据一致性。

预分配器

创建今后要用到的文档。

聚合器

更新文档内部的聚合数据，使之为最新数据。

其他有用的脚本（和本章联系不太紧密）如下。

译注1：对象文档映射，类似于关系型数据库的ORM。

结构校验器

确保当前所用的文档都有指定的字段，否则就自动校正或者发送通知。

备份任务

定期对数据库做 `fsync`、加锁和导出操作。

在后台执行一些检查和保护数据的脚本，能解除很多后顾之忧。

第 2 章

实现技巧

2.1 技巧14：使用正确的类型

用正确的类型存放数据大有裨益。数据类型影响数据的查询方式、MongoDB 的数据存放顺序，以及占用多少空间。

数字

作为数字使用的字段都应该作为数字存储。也就是要做计算或者按大小排序的字段。但是，用哪种数字呢？有时无所谓，有时则要慎重考虑。

按大小排序所有数字类型都没问题。比如把一个 32 位整数 2、64 位整数 1 和一个双精度浮点数（64 位浮点数）1.5 排序，不会有任何问题。但是一些特定操作需要数据为某些特定类型，如位操作（AND 和 OR）只适用于整数（不能是双精度浮点数）。

数据库会自动转换溢出的（比如由于 `$inc` 操作导致的溢出）32 位浮点数，将其变成 64 位整数，所以不必多虑。

日期

和数字类似，是精确的日期就要存成日期的类型。但是，有些日期（比如生日）并不是很精确，谁也不会知道自己出生时间的毫秒数。诸如此类的日期，用 ISO 格式的日期就好了，也就是 *yyyy-mm-dd* 形式的字符串。这样对生日排序也不会有问题，用起来更方便，而使用 `date` 类型总会要求匹配到毫秒级别。

字符串

MongoDB 中的字符串都是 UTF-8 编码的，所以其他编码形式的字符串必须要么转换成 UTF-8 编码，要么以二进制形式存储。

`ObjectId`

`ObjectId` 就要作为 `ObjectId` 存储，千万不要存成字符串。这点非常重要，原因如下。第一，方便查询（字符串和 `ObjectId` 不能相互匹配）。第二，`ObjectId` 含有有用的信息，绝大多数驱动都有方法从 `ObjectId` 中获知文档的创建日期。第三，字符串表示的 `ObjectId` 要多占两倍磁盘空间。

2.2 技巧15：用简单唯一的id替换_id

要是数据本身没有一个唯一的字段（通常就是这样），那么就用默认的 `ObjectId` 作为 `_id`。但是，若是数据本身就有唯一的字段，并且不需要 `ObjectId` 的功能，那

么就用自己唯 一的值覆盖掉默认的 _id 好了。这样会节省一些空间，尤其是要对该字段创建唯一索引时，就会省下一个索引的空间和资源（非常显著的节约）。[1]

使用自定义的 _id 还有些弊端。第一，要确保其唯一性，并且要处理键值重复异常。第二，要留意索引的树结构（技巧 22），要清楚随机和非随机的插入顺序会怎样。就索引树而言，ObjectId 的插入顺序非常好，因为它的值总是增加的，所以插入总是在 B 树的右侧边上进行。这样 MongoDB 只需要将 B 树的右侧边放在内存中就可以了。

与之相反，_id 中随机的值会导致在整个树上插入。这样系统就必须将对应的索引页面调入内存，更新一点点，然后就可能放之不管，直到它被系统移出内存。这样效率不高。

2.3 技巧16：不要用文档做_id

除了不可避免的情况（如 MapReduce 的输出），通常都不应该将文档作为 _id。问题就在于索引一个文档中的字段和索引文档完全不一样，如果没有每次查询整个子文档的计划，最后会有多个索引，如 _id、_id.foo、_id.bar 等。

更改 _id 必须得覆盖整个文档，所以若子文档的字段有变化则更新非常不便。

2.4 技巧17：不要用数据库引用

> 这个技巧特指子文档形式的数据库引用，不是通常意义的引用（如前一章所述）。

数据库引用一般是形如 {$id : identifier, $ref : collectionName}（也可以有可选的 $db 字段，表示数据库名）的常见子文档。这有点像关系型数据库，引用了另一个集合中的文档。但是，并不是真正引用了其他集合，仅仅是一个普通的子文档。完全没有什么神奇之处。MongoDB 也不能在必要的时候即时解除数据库引用。MongoDB 也不用这种方式做联结。它们只是存放了 _id 和集合名的普通子文档。也就是说为了解除数据库引用必须要对数据库额外做一次查询。

若被引用文档的集合是确定的，不妨只用 _id 引用，这样比同时用 _id 和集合名要节省空间。如果知道要引用的集合，数据库引用就显得有些浪费空间。

译注1：_id自带了唯一索引，且不可取消。

我只听说过一次数据库引用产生了积极地效果，那个系统中的用户可以评论系统中的任何事情。`comments` 集合中存储着数据库引用，这些引用几乎覆盖了系统的每个角落。

2.5　技巧18：不要用GridFS处理小的二进制数据

GridFS 需要查询两次，一次获取文件的元信息，另一次获取其内容（图 2-1）。所以如果用 GridFS 存储小文件，会使应用查询次数加倍。从根本上说，GridFS 是用来将大的二进制对象切成小片存放到数据库中。

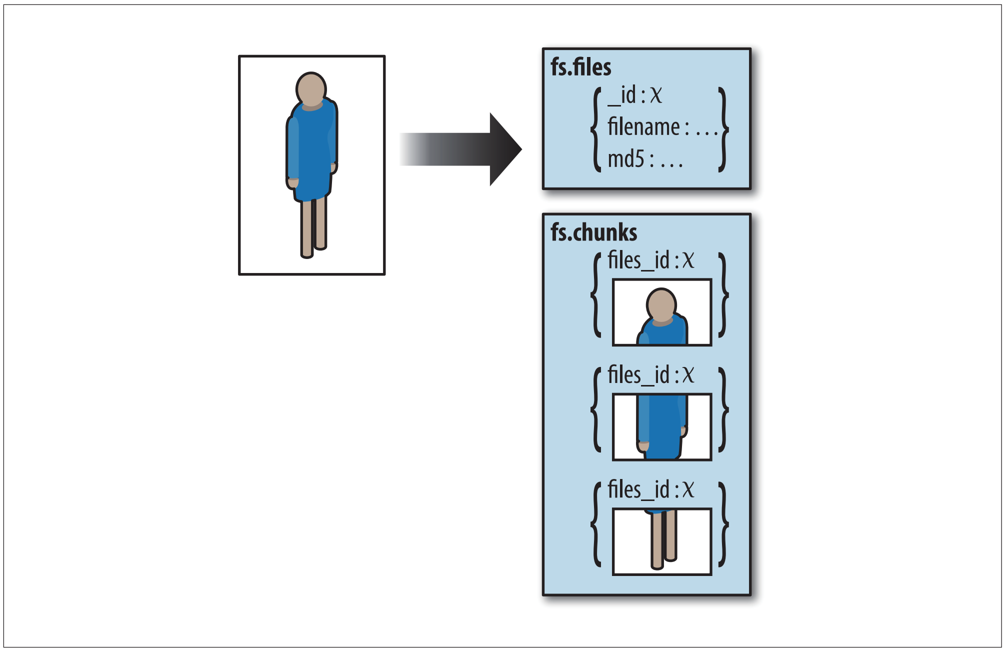


图 2-1：GridFS 将大数据分成小块存储

GridFS 是用来存放大数据的——至少得是一个文档放不下的数据。根据经验来讲，因为过大而使客户端一次加载不了的东西一般也不需要服务端一次加载。所以，那些可以作为数据流传递给客户端的数据非常适合应用 GridFS。需要在客户端一次全部加载的东西，如图片、声音，甚至小的视频，通常都应该嵌到主文档中。

延伸阅读：

- GridFS 机理（http://www.mongodb.org/display/DOCS/GridFS）。

2.6 技巧19：处理“无缝”故障切换

人们经常听到 MongoDB 可以做无缝的故障切换，所以当遇到异常就会非常惊讶。MongoDB 不需要干预就会尝试进行故障恢复，但是有些特定错误是它无法自动处理的。

假设发送给服务器一个请求，得到了网络错误。这时驱动有很多选择。

若驱动不能重连数据库，显然就不能重新尝试发送请求。但是，若驱动知道有另一台服务器，它可以自动对其发送请求吗？这要看到底是什么样的请求。如果是对活跃节点的写入，可能没有别的活跃节点了。要是读取，比如需要执行很久的 MapReduce 操作，本来执行任务的从属节点出故障了，驱动也不应该转发请求到随便选的其他服务器（万一是活跃节点呢？）。所以，对另一台服务器不能自动重试。

要是错误仅仅是临时的网络错误，且驱动立刻重新获得了服务器的网络连接，它也不应该重新发送请求。要是驱动发送原始请求后发生网络故障，或者在服务器响应的时候发生故障该怎么办？数据库可能已经处理了请求，所以不必再次发送。

这个令人头痛的问题总是和应用相关，所以驱动将问题推给了使用者。用户必须要处理网络异常（详见驱动文档）。处理异常，然后逐请求地判断要重新发送请求吗，要先确定数据库状态吗，可以直接放弃吗，还是需要继续重试？

2.7 技巧20：处理复制组失效及故障恢复

应用要能处理所有复制组会遇到的极端情况。

假设应用抛出了“not master”异常。可能有几个原因：复制组正在做故障恢复，应用必须要平滑地应对活跃节点选举的这段时间。选举的时间不尽相同，一般只需要几秒钟，但也有超过 30 秒的时候。若是被分割到了网络上状况不佳的一部分，可能数小时都连不上主节点。

看不到主节点是要重点处理的情况，如果遇到了这种情况，你的应用能降级为只读模式吗？应用要能应对短时（活跃节点选举）只读和长时（大部分机器宕机或者网络割裂）只读。

无论有没有活跃节点，都应该能继续从能连接的节点读取数据。

节点在选举期间会进入短暂的不可读的“恢复”阶段，驱动若是请求读取处于这种状态的节点，则这些节点会抛出错误表明自己“既不是主节点也不是从属节点”。这种状态也可能非常短暂，以至于就被驱动发送给数据库的两次 ping 错过了。

第 3 章

优化技巧

编写应用的人通常都希望数据库能即时响应。为了使应用尽可能做到这点，就必须要知道什么最耗时间。

3.1 技巧21：尽可能减少磁盘访问

内存访问比磁盘访问要快得多。所以，很多优化的本质就是尽可能地减少对磁盘的访问。

模糊数学

内存的读取速度比磁盘快一百万倍。

比方说通过机械硬盘访问数据要 10 毫秒，而内存访问只要 10 秒。（这在很大程度上取决于硬盘和内存的种类，此处只是笼统的说法，力求给人一个大致的印象。）对应的时间比值就是 1 毫秒比 1 纳秒。1 毫秒等于 100 万纳秒，所以磁盘访问时间要比内存长大约一百万倍。

所以从计算结果得知，读磁盘要消耗很长时间。

> 在 Linux 中，我们可以通过运行 `sudo hdparm -t /dev/hd`*`whatever`* 来查看连续磁盘访问的信息。但不会很精确，因为 MongoDB 还有些非连续的读写，不过看看机器能做什么也挺有趣的。

那么，究竟该如何应对呢？有几种“简单”的办法。

使用 SSD

SSD（固态硬盘）在很多情况下都比机械硬盘快很多，但容量小，价钱高，难以安全清除数据，与内存读取速度的差距依旧明显。但是，还是可以尝试用用的。一般来说 SSD 与 MongoDB 配合得非常完美，但也不是一定见效的灵丹妙药。

增加内存

增加内存可以减少对硬盘的读取。但是，增加内存也只能解决燃眉之急，总有内存装不下数据的时候。

所以，问题就变成了如何在磁盘上存放太字节级别（拍字节级别？）的数据，而同时让应用尽可能只读取在内存中的数据，又尽量避免磁盘和内存之间的数据移动。

若要真正实时且随机地访问数据，就只能仰赖大量内存了。但是绝大多数应用并不

是这样，访问新数据比老数据更频繁，一些用户比其他用户更加活跃，特定区域比其他地方拥有更多的客户。这类应用可以通过精心设计，让一部分文档在内存中，极大减少硬盘访问。

3.2　技巧22：使用索引减少内存占用

首先，看看图 3-1，其中展示的是读请求的处理顺序。

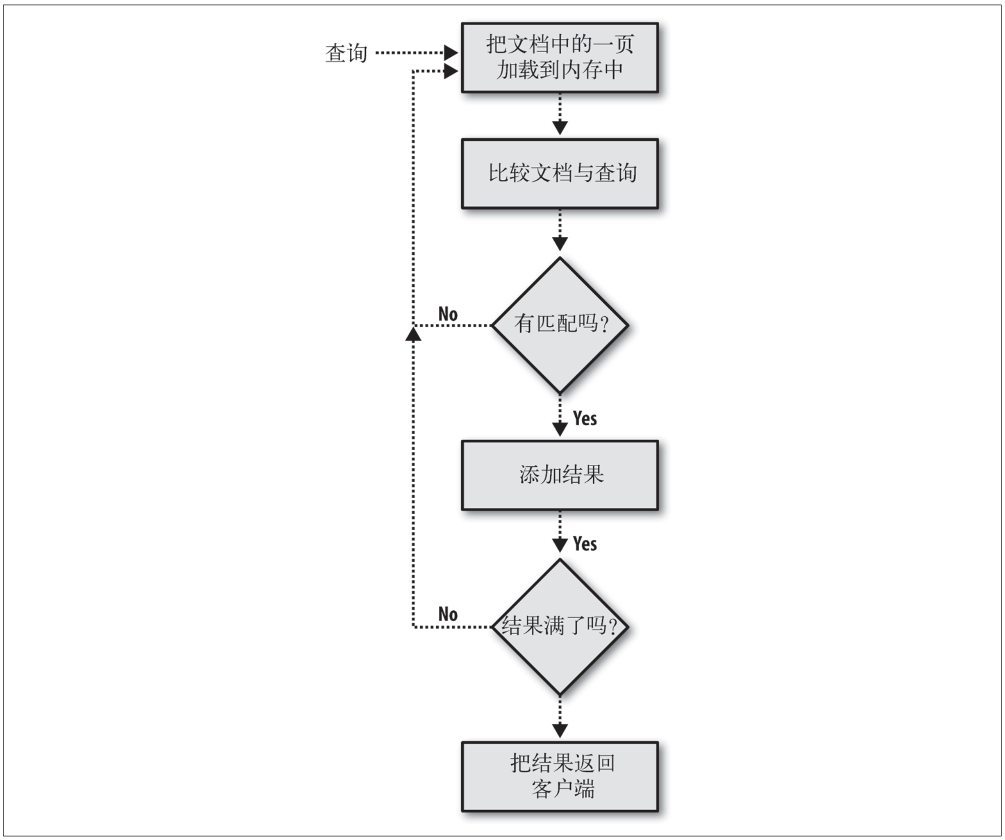


图 3-1：对数据库请求的过程

本书中假设内存页面大小为 4 KB，当然实际中并不总是如此。

假设机器共有 256 GB 的数据，16 GB 的内存，并且几乎所有数据都在一个集合上，现在要查询这个集合。MongoDB 会如何处理呢？

MongoDB 将文档的第一个页面加载到内存并与查询比较。然后加载下一页并比较。如此往复，直至比较完全部数据。没有捷径可循，不瞧一瞧文档怎么能得知到底能

不能匹配呢？所以必须要遍历整个集合。因此，需要全部加载 256 GB 数据到内存（当需要空间时操作系统会自动将旧的页面换出去）。整个过程相当漫长。

怎样避免每次查询都加载全部数据的情况呢？可以让 MongoDB 对指定的字段 *x* 创建索引。MongoDB 会创建相应的树，大体上预处理这些数据，并将这一集合中的每个 *x* 值都添加到有序树中（参见图 3-2）。树中每个索引项都含有 *x* 的值和一个指向对应文档的指针。树中只有指向文档的指针，而不是文档本身，也就是说索引要比整个集合小很多。

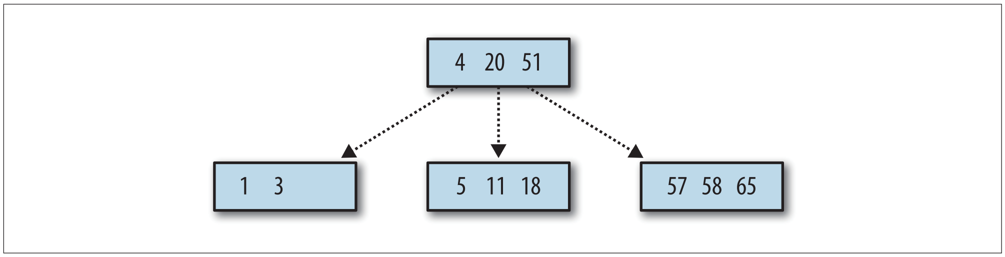


图 3-2：B 树，可能对应一个整数字段的索引

当查询条件中有 *x*，MongoDB 会发现并利用 *x* 的索引。这样就不必查找每个文档，MongoDB 会这样判断："我要找的值大于还是小于树中当前节点的值？若大于，则去右侧，若小于，则去左侧。"如此往复，直到找到欲查询的值或者发现其不存在。若是找到了相应值，则按照对应的指针把文档页面加载到内存，并返回结果（参见图 3-3）。

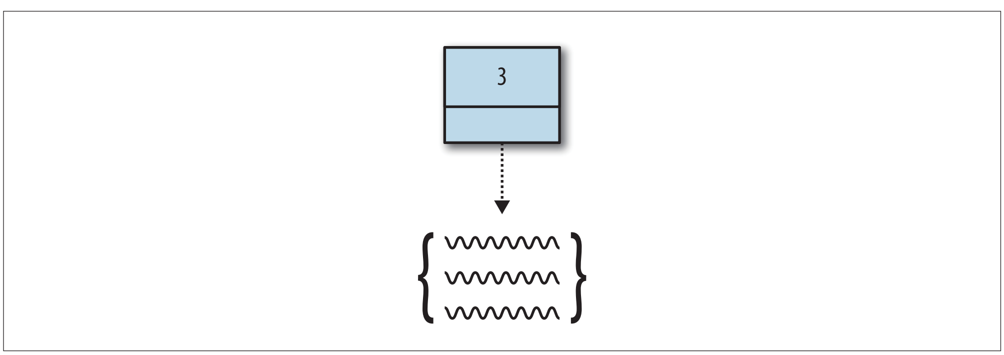


图 3-3：索引项包含索引的值和到对应文档的指针

所以，假设查询能在集合中找到一两个匹配的文档。不用索引的话，我们必须把 64 000 000 个页面从硬盘加载到内存。

页面：256 GB / (4 KB/ 页面) = 64 000 000 页面

若是索引有 80 GB，则索引有 20 000 000 页面。

页面：80 GB / (4 KB/ 页面) = 20 000 000 页面

另外，索引是有序的，所以不必遍历全部项，只要加载其中的一部分节点就可以了。有多少呢？

要加载进内存的页面：ln(20 000 000) = 17 页面

注意，64 000 000 竟然减少到了 17！

诚然，17 并不准确，找到索引中的结果后还要把文档加载到内存，所以文档大小也得计算在内，另外树中的节点也可能超过一个页面大小，也要额外计算。但不管怎么说，页面的总数与遍历整个集合比起来还是微不足道的。

希望大家现在对索引加速查询的机理有个大概的认识了。

3.3 技巧23：不要到处使用索引

上面的介绍使你惊叹于索引的强大作用，但要提醒你，不是所有查询都可以用索引的。比如，在刚才的例子中，要是需要返回集合中 90% 的文档而非获取一些记录，就不应该用索引。

如果对这种查询用了索引，结果就是几乎遍历整个索引树，把其中一部分，比方说 60 GB 的索引都加载到内存。然后按照索引中的指针加载集合中 230 GB 的文档数据。最终将加载 230 GB + 60 GB =290 GB，比不用索引还多。

所以，索引一般用在返回结果只是总体数据的一小部分的时候。根据经验，一旦要大约返回集合一半的数据就不要使用索引了。

若是已经对某个字段建立了索引，又想在大规模查询时不使用它（因为使用索引可能会较低效），可以使用自然排序，用 `{"$natural" : 1}` 来强制 MongoDB 禁用索引。自然排序就是“按照磁盘上的存储顺序返回数据”，这样 MongoDB 就不会使用索引了：

```
> db.foo.find().sort({"$natural" : 1})
```

如果某个查询不用索引，MongoDB 会做*全表扫描*，即逐个扫描文档，遍历整个集合，以找到结果。

写入速度

每当增加、删除、更新记录，所有相应的索引也必须更新。插入文档时，MongoDB

需要找到文档中的值在每一个索引树中的位置，然后在那儿插入。删除时，要找到树中的索引项并删除。更新时，也可能像插入时那样新建索引项，也可能像删除时那样删除索引项，若是值更新了就会既有添加又有删除。所以，索引会增加很多额外的写入。

3.4 技巧24：索引覆盖查询

如果只想返回某些字段且所有这些字段都可放在索引中，MongoDB 可以做索引*覆盖查询*（covered index query），这种查询不会访问指针指向的文档，而是直接用索引的数据返回结果。比如，有如下索引：

```
> db.foo.ensureIndex({"x" : 1, "y" : 1, "z" : 1})
```

现在查询被索引的字段，并只要求返回这些字段，MongoDB 就没必要加载整个文档：

```
> db.foo.find({"x" : criteria, "y" : criteria},
... {"x" : 1, "y" : 1, "z" : 1, "_id" : 0})
```

这样查询仅仅访问了索引的数据，而没有访问整个集合的数据。

注意结果子句 `"_id":0`，`_id` 是默认返回的，但不是上面索引的一部分，所以 MongoDB 就需要到文档中获取 `_id`。将其去掉，MongoDB 就可以仅根据索引返回结果了。

若是查询只返回几个字段，则考虑将其放到索引中，即使不对它们执行查询，也能做索引覆盖查询。例如，上面的查询中并没有用到 `z`，但是是个要返回的字段，所以放到了索引中。

3.5 技巧25：使用复合索引加快多个查询

若有可能，要建立能被多个查询利用的复合索引。这未必能实现，但当多个查询的参数相似时就有机会。

查询只要和索引开头部分匹配就能利用索引。所以，建立索引时要考虑这些查询依赖的所有字段。

假设应用需要执行这些查询：

```
collection.find({"x" : criteria, "y" : criteria, "z" : criteria})
collection.find({"z" : criteria, "y" : criteria, "w" : criteria})
collection.find({"y" : criteria, "w" : criteria})
```

可以看到，y 字段是唯一一个每个查询都有的，所以很适合放到索引中去。z 在前两个中出现了，w 在后两个中出现了，因此它们两个都是仅次于 y 字段的合适选项（索引排序详见技巧 27 以及技巧 28）。

索引的命中率越高越好。某个查询如果更重要或者执行更频繁的话，索引也要针对它进行优化。例如，假设第一个查询的执行次数是后两个的数千倍，则选择对其建立索引：

```
collection.ensureIndex({"y" : 1, "z" : 1, "x" : 1})
```

这样就是针对第一个查询尽可能做了优化，后两个查询能部分利用这个索引。

若是 3 个查询执行的情况大体相当，下面的索引则更佳：

```
collection.ensureIndex({"y" : 1, "w" : 1, "z" : 1})
```

这样 3 个查询都能利用索引做 y 条件的查询，后两个查询能用到 w 的部分，中间的查询则可以完全利用这个索引。

可以用 explain 来查看查询中是如何使用索引的：

```
collection.find(criteria).explain()
```

3.6 技巧26：通过建立分级文档加速扫描

将数据组织得有层次，不仅可以让其看着更有条理，还可让 MongoDB 在（偶尔）没有索引时也能快速查询。

例如，假设有个查询并不使用索引。如前文所述，MongoDB 需要遍历集合中的所有文档，来确定是否有什么能匹配查询条件。这个过程可能相当耗时，而这取决于文档结构。

比方说，用户文档使用了扁平的结构：

```
{
    "_id" : id,
    "name" : username,
    "email" : email,
    "twitter" : username,
    "screenname" : username,
    "facebook" : username,
    "linkedin" : username,
    "phone" : number,
    "street" : street,
    "city" : city,
```

```
    "state" : state,
    "zip" : zip,
    "fax" : number
}
```

假设我们要执行下面的查询：

```
> db.users.find({"zip" : "10003"})
```

MongoDB 会做哪些工作呢？它必须遍历每个文档的每个字段，来查找 zip 字段（图 3-4）。

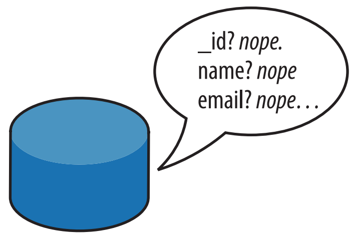


图 3-4：若文档没有层次的话，MongoDB 必须遍历文档中的每个字段

使用内嵌文档我们可以建立自己的“树”，能让 MongoDB 执行比此查询时更快。现在结构变了：

```
{
    "_id" : id,
    "name" : username,
    "online" : {
        "email" : email,
        "twitter" : username,
        "screenname" : username,
        "facebook" : username,
        "linkedin" : username,
    },
    "address" : {
        "street" : street,
        "city" : city,
        "state" : state,
        "zip" : zip
    }
    "tele" : {
        "phone" : number,
        "fax" : number,
    }
}
```

相应地查询也变了：

```
> db.users.find({"address.zip" : "10003"})
```

这样 MongoDB 在找到匹配的 address 之前，仅查看 _id、name 和 online，而后在 address 中匹配 zip。合理使用层次可以减少 MongoDB 对字段的访问。

3.7　技巧27：AND型查询要点

假设要查询满足条件 A、B、C 的文档。若满足 A 的文档有 40 000，满足 B 的有 9 000，满足 C 的有 200。要是让 MongoDB 按照这个顺序查询，效率可不高（参见图 3-5）。

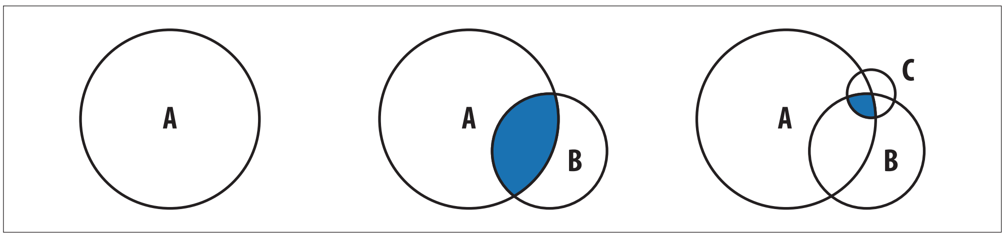


图 3-5：深色部分表示每步都必须搜索的查询空间、相比较而言，按照结果数量由大到小的顺序进行的查询多做了好多

如果把 C 放在最前，然后是 B，最后是 A，则针对 B 和 C 只需要查看（最多）200 个文档（图 3-6）。

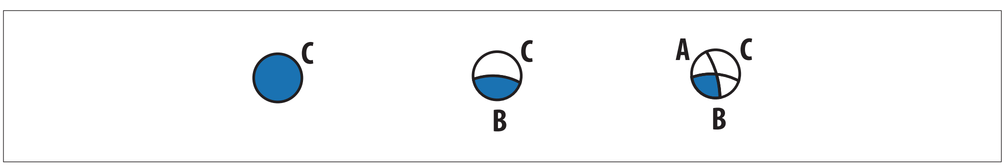


图 3-6：将条件 C 放在最前面，后续查询的搜索空间将明显减少

这样工作量显著减少了。要是已知某个查询条件更加苛刻，则要将其放置在最前面（尤其是在它有对应索引的时候）。

3.8　技巧28：OR型查询要点

OR 型查询与 AND 型查询正相反，匹配最多的查询语句应该放在最前面，因为 MongoDB 每次都要匹配不在结果集中的文档。

若是按照 AND 型的顺序查询，每次都要查询很多文档（图 3-7）。

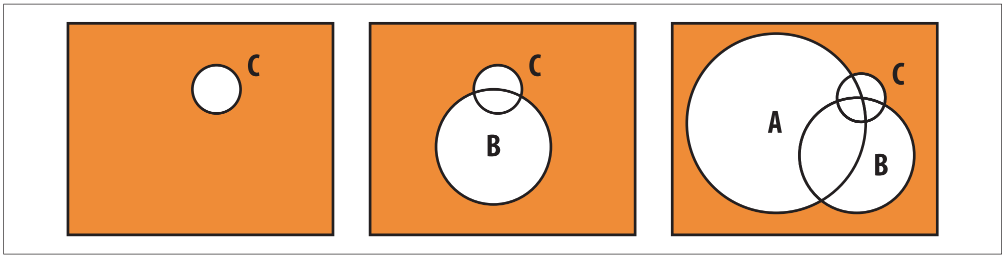


图 3-7：矩形表示整个集合，深色部分表示每一步都要搜索的空间。若先按 C 进行搜索，则后续的每步都需要查绝大部分的集合

因此，我们应将匹配最多的放在最前面（图 3-8）。

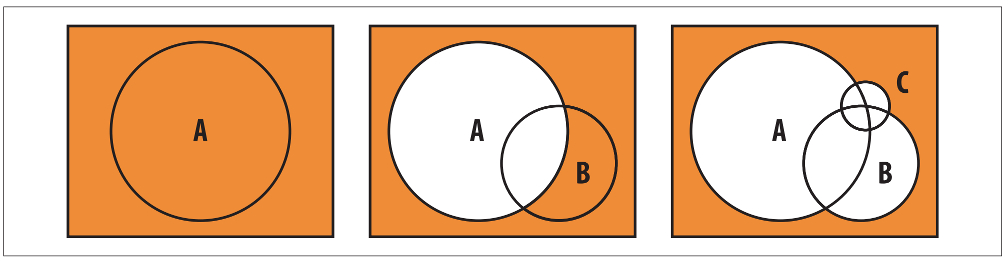


图 3-8：将结果最多的条件前置能缩小后续查询的搜索空间

第 4 章

数据安全性和一致性

4.1 技巧29：单机做日志，多机则复制

理想情况下，所有的写入都能立刻执行并永久地保留在磁盘上，可以被立刻访问到。可惜现实情况并不是这样，要么需多花时间确保数据安全，要么只能加快速度但使数据不怎么安全。这是 MongoDB 灵活性的最佳体现，所以一定要掌握各种方法。

复制和日志是 MongoDB 确保数据安全性的两种途径。

通常都要使用复制，而且至少要有一台服务器启用日志。MongoDB 博客有篇很棒的文章，说明了为什么不应该在单个服务器上运行 MongoDB（别的数据库也一样）。

MongoDB 的复制自动在服务器间同步写入操作。如果活跃节点停机，则可以将另一台服务器选作主节点（如果用了复制组，这个过程就是自动的）。

如果复制组中的一个节点没有正常停机，也没有启动 `--journal` 选项，MongoDB 并不能保证数据安全性（数据可能已损坏）。所以得有一份干净的数据，无论删掉后重新同步，还是加载备份并快速同步（http://www.mongodb.org/display/DOCS/Upgrading+to+Replica+Sets#UpgradingtoReplicaSets-UsingsSlave%27sExistingData）。

日志能保证单个服务器的数据安全。所有操作都将被记录下来（日志）并定期写入磁盘。如果机器崩溃了，但硬盘还是好的，你就可以重启服务器，数据会自动根据日志完成修复。记住要是硬件出了问题，MongoDB 也无能为力。如果硬盘坏了，数据库很可能恢复不了。

虽然复制和日志可以同时使用，但是性能会受到影响。两种方式都会将所有写入进行备份，所以你可以进行如下选择。

无安全措施

每个请求写入一次。

复制

每个请求写入两次。

日志

每个请求写入两次。

复制 + 日志

每个请求写入 3 次。

将一条信息写入 3 次确实有点多，但是要是性能无需太高，恰恰数据安全性却非常重要，没准可以两者结合使用。后面会讲一些既安全又更高效的方案。

4.2　技巧30：坚持使用复制或日志,或两者兼用

单一服务器要用 `--journal` 选项。



开发过程中建议一直开着 `--journal` 选项。在本地 MongoDB 配置中开启日志是为了确保开发过程中不丢失数据。

由于使用日志会导致性能下降，多台机器情况下可以混合使用启用日志的和不启用日志的服务器。备份用的从属节点可以用日志，活跃节点和热备节点（尤其是做读取负载均衡的节点）可以不用。

图 4-1 展示了一组小巧健壮的设置。活跃节点和热备节点都没有开启日志，这样能够快速读写。通常遇到服务器崩溃时，可以切换到热备节点，然后有空的时候再来重启坏掉的服务器。

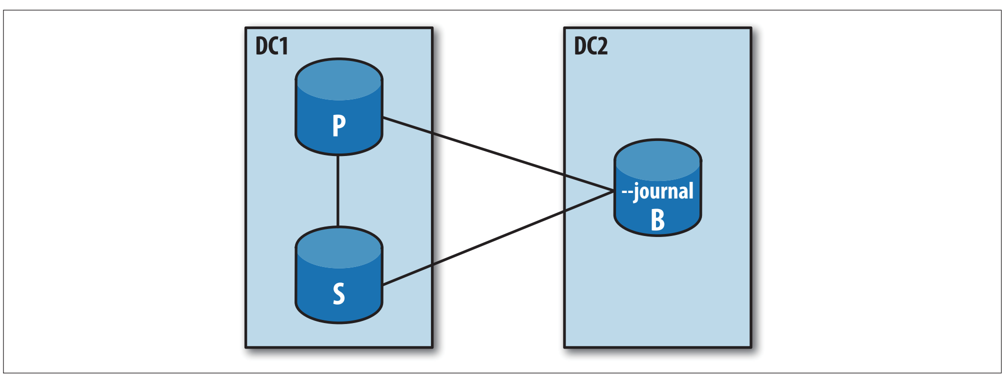


图 4-1：活跃节点（P）、热备节点（S）和开启日志的备份服务器

即便其中一个数据中心完全停止运行，数据依然是安全的，因为还有一个安全的数据副本。DC2 停机时，如果又重新开始运转，还是重启备份服务器就好了。如果 DC1 停机，可以将备份机器切换成主节点，或者用其数据恢复 DC1 的机器。就算两个数据中心都停机，至少 DC2 的备份数据能让我们用于引导恢复。

图 4-2 展示了 5 台机器的安全设计方案。这个方案比前一个方案更可靠，两个数据中心都有热备节点，还有一个加了时间延迟用来预防用户误操作，还有一台为了以防万一启用了日志。

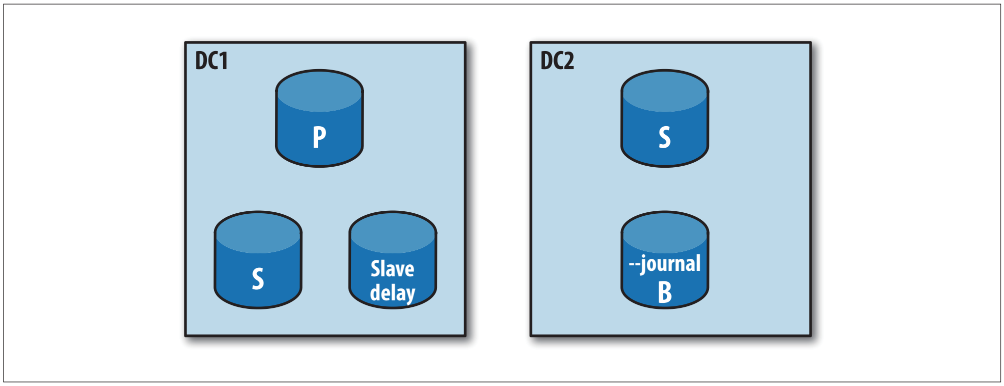


图 4-2：一个活跃节点（P），两个热备节点（S）一个有时间延迟的从属节点，以及一个开启日志的备份服务器（B）

4.3 技巧31：不要信任repair恢复的数据

如果数据库溃了，且没有启用 --journal 选项，千万不要将这些数据拿来就用。可能几个星期都平安无事，但突然间访问到了损坏了的文档，应用程序就遭殃了。另外，也可能由于索引的混乱导致返回的结果不完整，还可能导致其他诸多问题。崩溃导致的问题比较严重，而且可能潜伏且通常很长一段时间都不能被发现。

当然还是有些选择的。运行 repair（http://www.mongodb.org/display/DOCS/Durability+and+Repair#DurabilityandRepair-RepairCommand）是一种诱人的做法，但却是个最不适当的选择。首先，这个过程要遍历所有能找到的文档并复制。这个过程极其耗时，而且需要大量磁盘空间（至少与现在使用的空间相同），而有问题的文档会被直接忽略掉。要是因为崩溃，数以万计的文档找不到的话，就不能被复制，也就意味着丢失了。数据库是正常了，但数据却可能丢很多。此外，repair 的修复也是有限的，它并不能知晓文档的细节，所以若崩溃导致某些字段无法解析，repair 是无法发现或修复的。

我们比较推荐的做法是从备份快速恢复，或者从头开始同步。注意，在同步数据之前一定要清除所有损坏的数据，MongoDB 的复制是不能修复这些损毁数据的。

4.4 技巧32：getlasterror

默认情况下，写入是不返回任何数据库响应的。也就是说若发送给数据库更新、插入、删除这样的指令，数据库在执行完成后不返回给用户任何确认信息。所以，驱动也得不到指令是否成功执行的响应。

然而，很多情况下需要得到数据库的响应。为此，MongoDB 有个返回上一个指令执行状态的命令，叫做 getlasterror。起初，它只是用来描述上一个指令的错误信息，但后来扩展到了描述各种写入信息并提供了一系列与安全性相关的选项。

为了避免“读取最近写入”这种粗心的错误（详见技巧 50），getlasterror 会和写入请求捆绑在一块，强制数据库将两者作为一个请求。两者被同时发送并确保连续执行，期间没有任何别的操作。驱动会自动处理这些，所以使用者并不需要特别留意，就把它当做“安全”写入好了。

4.5 技巧33：开发过程中一定要使用安全写入

开发过程中，你都会希望应用程序的行为与预期一致，这就离不开安全写入了。写入都会导致哪些错误呢？某次写入可能会向非数组字段 push 数据，导致键重复异常（试图向有唯一索引的字段存放两个值相同的文档），删除一个 _id 字段，否则会有数百万其他用户错误。部署之前需要知道这些写入是否合理。

耗尽磁盘空间也是不太容易被发现的错误，比如突然间查询神秘“丢失”了一些数据。如果没有用安全写入，就不太容易发现，因为一般你也不会想到检查磁盘。我经常将 --dbpath 设置到错误的分区，导致 MongoDB 耗尽磁盘的时间比预期早得多。

在开发过程中，有许许多多的开发错误会导致写入失败，这些都是需要了解和处理的。

4.6 技巧34：使用w参数

有些重要的操作，需要将写入数据复制到复制组的大多数节点。直到大多数节点都写入了，写入才算提交完成。若是写入未能提交完成，同时网络发生割裂或者服务器由于故障与复制组的多数节点失去联络，写入就会回滚。（题外话：要是对回滚感兴趣，大家可以参考我的一篇博文，其中讲述了如何操作，地址为 http://www.snailinaturtleneck.com/blog/2011/01/19/how-to-use-replica-set-rollbacks/。）

w 表示至少要有多少台服务器写入了才返回成功。过程也简单，不过是发送 getlasterror 到服务器（通常就是针对某一具体的写操作设置 w）。服务器清楚自己在 oplog 中的位置（“在 123 号操作”），然后等待 w − 1 个节点完成 123 号操作。每当有从属节点完成指定操作，活跃节点就将 w 计数减 1。一旦 w 等于 0，getlasterror 就返回成功。

注意，复制的写入总是按照一定次序进行的，不同的节点可能处在不同的“历史位

置”，但数据集一定是一致的。它们会和活跃节点一分钟前，也可能是几秒钟前，或一周以前等完全一样。但绝不会丢失任何操作。

也就是说，以下命令能够强制 *num*-1 个节点与活跃节点同步：

```
> db.runCommand({"getlasterror" : 1, "w" : num})
```

所以，开发者会问：如何设置 w 呢？前面提到过，要想绝对安全就得大于节点数的一半。然而，小于半数也是可行的。

w 如果小于服务器数的一半，则更易于完成，可能已经够用了。若是这一小部分节点由于网络割裂或者服务器故障与复制组失去联系，则复制组中另外过半数的节点会选举新的活跃节点，并丢失已经同步到 w 个节点的操作。然而，只要有一个已经同步的节点与复制组依然保持联系，那么复制组中的其他节点都会在选举新活跃节点前同步这个写入操作。

将 w 设置为超过服务器数目的一半，如果网络发生割裂或者某些服务器停止运行，只有处理完写入的活跃节点才能被选举。这个条件为数据安全提供了有力保障，但不太好达成，因为需要越多节点同步，完成的可能性就越低。

4.7 技巧35：一定要给w设置超时

假设有一个含 3 个节点的复制组（一个活跃节点和两个热备节点），现在要做两个从属节点与主节点间的同步：

```
> db.runCommand({"getlasterror" : 1, "w" : 2})
```

但要是有个热备节点停机了怎么办？ MongoDB 不会检查集群内热备节点的数量，它会等待 w 个从属节点复制完成，可能是 2 个、20 个或者 200 个（要看具体的 w 了）。

所以，使用 getlasterror 时一定要合理设置 wtimeout 参数。wtimeout 表示从节点报告返回并失败时超时等待的毫秒数。下面以 100 毫秒为例：

```
> db.runCommand({"getlasterror" : 1, "w" : 2, "wtimeout" : 100})
```

注意 MongoDB 应用复制操作的顺序。如果将 A、B、C 写入到主节点，复制到从属节点的时候也是 A、B、C 这种顺序。假设现在的情况如图 4-3 所示。如果在主节点上写入 N，然后执行 getlasterror，从属节点必须将 E ~ N 都复制成功，而后 getlasterror 才能返回成功。所以，如果从属节点的复制慢了，getlasterror 会使应用运行速度变得很慢。

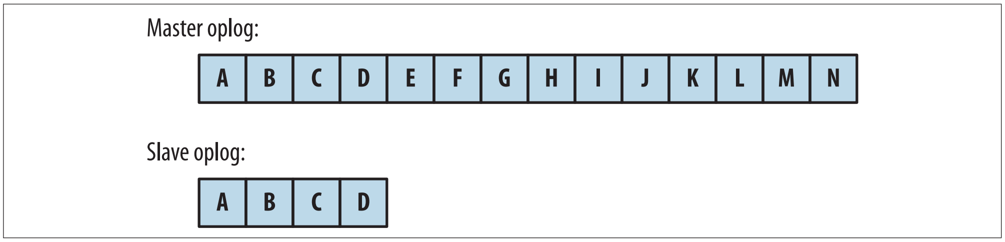


图 4-3：主节点和从属节点的 oplog。与活跃节点相比，从属节点落后了 10 个操作

还有一件事儿在掌控范围内，那就是如何处理 getlasterror 超时。显然，保证到另一服务器的复制是因为这个写入很重要。但要是写入本地没问题，但是不能复制到足够的备份机器，该怎么办呢？

4.8　技巧36：不要每次写入都调用fsync

要是有重要的数据需在日志中有记录，则写入时一定要使用 fsync 选项。fsync 会等待数据都成功写入日志(至多 100 毫秒)，然后才会返回成功。要注意的是，fsync 并不是立即将数据写入磁盘，而是让应用程序暂停直至数据被写入磁盘。所以，若每次插入都运行 fsync，则每 100 毫秒只能插入一次。这比 MongoDB 普通插入不知慢了多少倍，所以还是少用 fsync 为妙。

通常 fsync 只与日志搭配使用。除非特别清楚，否则绝不要在未开启日志时使用 fsync。不听忠告会得不偿失，严重影响应用性能的。

4.9　技巧37：崩溃之后正常启动

如果开启了日志，并且虽然崩溃，但系统可以恢复（比如硬盘没有受损，机器也没有进水），数据库可以正常重启。注意使用常规选项，尤其是 --dbpath（以便找到日志文件）和 --journal。MongoDB 会自动修复数据，而后才开始接受连接。数据比较多时这可能会需要几分钟，但比起 repair 要省很多时间了（大约 5 分钟左右）。

日志文件存放在日志目录中，千万不要删除。

4.10　技巧38：持久性服务器的瞬时备份

备份开启日志的数据库有两种方法，其一就直接对文件系统做快照，其二就是用 fsync 和锁配合，然后导出数据。注意不能没有执行 fsync 和上锁就直接复制所有文件，因为文件复制不能瞬时完成。若复制数据库和日志的时间点不同，这样还不如不备份。（一旦应用了这些日志，就有可能破坏数据文件。）

第 5 章

管理技巧

5.1 技巧39：手工清理块集合

GridFS 将文件内容保存在块集合中，默认的集合名是 fs.chunks。文件集合中的每个文档都会指向块集合中的一个或多个文档。所以经常要检查，看看有没有孤悬的块（就是没有和文件关联的块）。若数据库在保存文件的过程中异常关机（块写入后才会写 fs.files 文档），就可能出现这种情况。

检查块集合要选择流量较小的时段（因为会往内存中调入大量数据），步骤如下：

```
> var cursor = db.fs.chunks.find({}, {"_id" : 1, "files_id" : 1});
> while (cursor.hasNext()) {
... var chunk = cursor.next();
... if (db.fs.files.findOne({_id : chunk.files_id}) == null) {
...     print("orphaned chunk: " + chunk._id);
... }
```

这样就能将所有孤悬块的 `_id` 列出来。

在删除所有这些孤悬的块之前，要确保其不是正被写入文件的组成部分！你应当用 `db.currentOp()` 指令取消当前操作，之后看看 fs.files 集合查看最近的 `uploadDate`（上传日期）。

5.2 技巧40：用repair压缩数据库

技巧 31 中讲到了通常不建议用 `repair` 去恢复数据（除非迫不得已）的原因。不过 `repair` 用来压缩数据倒是正合适。[1]

> 但愿这个技巧早些过时，希望一旦在线压缩的 bug 得到修正就不需要这个技巧了。（参见 http://jira.mongodb.org/browse/SERVER-2120。）

`repair` 基本上就是 `mongodump` 加 `mongorestore`，通过这一过程将数据都整理出来，形成整洁的数据副本，移除文件的所有数据碎片。（当有大量删除或者更新而导致数据移动，集合中就会出现很多空洞。）`repair` 会以压缩形式重新插入数据。

使用 `repair` 还有些注意事项。

- 这样会阻塞操作，所以不要在活跃节点上执行。一般先在热备节点上执行，然后让活跃节点和热备节点交换角色，再在原来的活跃节点（现在是热备节点）上执行这个操作。

译注1：2.0系统在这方面有改进。

- 需要使用是原来数据库两倍的磁盘空间。（比方说，若有 200 GB 的数据，则还要至少 200 GB 的空闲磁盘空间才能运行 `repair`。）

很多人都会遇到的一个问题就是要通过 `repair` 处理的数据量太大，比如数据库可能有 500 GB，整个服务器的硬盘也就 700 GB。这种情况下，就只能用 `mongodump` 和 `mongorestore` 来进行“手工”恢复了。

例如，假设 *ny1* 主机上大部分磁盘空间都被占用了。数据库有 300 GB，而其所在服务器上总的磁盘空间也就 400 GB。但是我们还有 *ny2*，它也有同样的 400 GB 硬盘，但什么数据也没有。首先，如果 *ny1* 是主节点，则需要令其降级，然后执行 `fsync` 并加锁，这样能确保磁盘上其数据的一致性。

```
> rs.stepDown()
> db.runCommand({fsync : 1, lock : 1})
```

然后登录到 *ny2* 上并执行 :

```
ny2$ mongodump --host ny1
```

这就会将数据导出到 *ny2* 上的 *dump* 目录下。

上面 `mongodump` 的执行速度很可能会受到网络速度的限制。如果可以在物理上访问机器的话，加一块硬盘，做本地的 `mongodump`。

导出成功后就可以在 *ny1* 上恢复数据了，如下。

1. 停止 *ny1* 上的 *mongod* 进程。
2. 备份 *ny1* 上的数据文件（比如做 EBS 快照），以防万一。
3. 删除 *ny1* 上的数据文件。
4. 重启 *ny1*（现在没有数据）。如果其在某个复制组中，启动时要指定一个不同的端口并且去掉 `-- replSet` 参数，为的是不（与其他复制组中的节点）混淆。

最后，在 *ny2* 上执行 `mongorestore`：

```
ny2$ mongorestore --host ny1 --port 10000 # specify port if it's not 27017
```

这样 *ny1* 就有压缩形式的数据库文件了，之后就可以正常启动了。

5.3 技巧41：不要改变复制组成员投票的权值

要是希望某个机器优先成为活跃节点，需要设置权值。1.9.0 中可以为某个节点设置相对较高的权值，这样此节点总会优先成为活跃节点。但是在 1.9.0 以前的版本中，

这个权值只能为 1（可以成为活跃节点）或者 0（不能成为活跃节点）。在 1.9.0 之前的版本中，唯一能让某个节点总成为活跃节点的方法就是将其他节点的权值都设为 0。

人们总是将服务器依据权值成为活跃节点这一件事与社会中的投票选举等同起来，认为可以通过增加某个节点的权值来让其赢得选举。但是，服务器一点也不“自私”，未必会投自己的票。复制组中的节点都是“公正无私”的，对自己和友邻都是完全同等对待的。

5.4 技巧42：无活跃节点时可重置复制组

如果复制组中只有少数节点在运行，系统就会忽略本地数据库，然后重新配置复制组。对于大多数情形而言，这样做没什么问题，但是这样会有一些停机时间，因为要重建复制组和重新分配 oplog。如果想让应用保持运行（但因为没有活跃节点，所以它只能是只读的），可以，但前提是得有多于一个从属节点仍在工作。

选一个从属节点，将其关停，然后去掉 `--replSet` 选项并用另外一个端口启动。例如，若原本是这样启动的：

```
$ mongod --replSet foo --port 5555
```

可以这样重启：

```
$ mongod --port 5556
```

这样复制组中的其他节点就不会将其作为这一复制组的一员了（因为端口不同了），它本身也将不会使用复制组的配置（因为根本就没告诉它说它曾是复制组的一员）。此刻，它就仅仅是个普通的 *mongod* 服务器。

接下来要更改复制组的配置信息，因此要通过 shell 连接这个服务器。切换到本地（*local*）数据库，将复制组的配置信息存到一个 JavaScript 变量中。例如，假设复制组中有 4 个节点，看上去可能如下：

```
> use local
> config = db.system.replset.findOne()
{
    "_id" : "foo",
    "version" : 2,
    "members" : [
        {
            "_id" : 0,
            "host" : "rs1:5555"
        },
```

```
        {
            "_id" : 1,
            "host" : "rs2:5555",
            "arbiterOnly" : true
        },
        {
            "_id" : 2,
            "host" : "rs3:5555"
        },
        {
            "_id" : 3,
            "host" : "rs4:5555"
        }
    ]
}
```

要更改配置就要将 config 对象改成我们期望的配置并且将其标记为“最新的”，这样集群就能自动更新。

上面的配置是针对 4 个节点的集群的，但现在假设要将其改成只有 3 个节点，包括 *rs1*、*rs2* 和 *rs4*。为此要把 *rs3* 从数组中删除，用 JavaScript 的 slice 函数就能完成：

```
> config.slice(2, 1)
> config{
    "_id" : "foo",
    "version" : 2,
    "members" : [
        {
            "_id" : 0,
            "host" : "rs1:5555"
        },
        {
            "_id" : 1,
            "host" : "rs2:5555",
            "arbiterOnly" : true
        },
        {
            "_id" : 3,
            "host" : "rs4:5555"
        }
    ]
}
```

一定不要把 *rs4* 的 _id 改为 2。这样会造成混乱。如果想向集群中添加节点，建议用 JavaScript 的 push 函数添加 _id 为 4、5 之类的条目。要是想既添加又删除节点，可能有点乱，用 JavaScript 函数 splice 就好了（也可以用 push 和 slice）。

然后增加版本号（config.version），以此来通知其他节点配置有更新，需要同步。

现在要再次确认配置文档。若把这配置搞坏了，可能会把整个复制集的配置彻底搞

坏。说的明白点，数据没什么危险，但可能得将所有服务器都关掉，并删除每个服务器上的本地数据库。所以要确保这个配置指向了正确的服务器，所有的 _id 都没有变，也没有把仲裁者和普通的节点搞混。

当确信配置完全符合预期时，就可以停掉服务器了。然后用正常参数重新启动就好了（--replSet 和标准的端口）。几秒钟后，其他节点就会对其发起连接，更新自己的配置，然后选出新的活跃节点。

延伸阅读：

- slice 函数（参见 http://www.w3schools.com/jsref/jsref-slice-array.asp）；
- push 函数（参见 http://www.w3schools.com/jsref/jsref-push.asp）；
- 复制组选项 (参见 http://www.mongodb.org/display/DOCS/Replica+Set+Configuration#ReplicaSetCOnfiquxation-TheReplicaSetConfigObject)。

5.5 技巧43：不必指定--shardsvr和--configsvr参数

依据文档来看，似乎配置分片必须设置这些参数，但事实并非如此。这些参数大体上只是改变端口而已（这个端口会严重扰乱已有的复制组），--shardsvr 将端口改为 27018，--configsvr 将端口改为 27019。如果在多台机器上设置多个服务器，这样能降低互联的难度，使所有 mongo 进程都运行在 27017 上，使所有分片都运行在 27018 上，使所有配置服务器都用 27019。从零开始创建集群时，这样设置可以极大地帮助了解各种情况，但将已有的复制组切换到分片也没必要太过担心。

--configsvr 不仅更改默认端口，还会开启 *diaglog*，它用来以可重放的形式记录配置数据库的所有操作，以防不测。如果用的版本是 1.6，则应用 --port 27019 和 --diaglog 两个参数来达到同样的目的，因为只有 1.6.5 以上版本的 --configsvr 选项才会开启 diaglog。要是使用版本 1.8，要用 --port 27019 和 --journal (而不是 --diaglog)。日志和 diaglog 效果差不多，但性能好很多。

5.6 技巧44：开发时才用--notablescan

MongoDB 有个 --notablescan 选项，一旦开启，就会在某个查询要做表扫描时返回错误（处理使用索引的查询时一切正常）。开发时想确保所有查询都用到索引时，这个选项就非常有用了，但不建议在生产环境中使用。问题是很多简单的管理任务都需要表扫描。如果已经带着 --notablescan 参数开启了 MongoDB，还想看看数

据库中集合的清单，很遗憾，需要做表扫描。想要做些管理更新，其中需要用到没有索引的字段？这可不行，不允许表扫描。

`--notablescan` 是调试的好工具，但只用索引查询通常极为不切实际。

5.7 技巧45：学习JavaScript

即便你用的语言有很好的 shell（比如 Python），或者有很好的抽象层把应用和 MongoDB 隔离开（比如 Mongoid），也应该熟悉 JavaScript shell。这种信息存取方式最快捷，且 Java Script 是所有 MongoDB 开发者通用的语言。

要尽可能充分地利用 shell，了解一些 Java Script 知识是很有帮助的。下面的这些技巧涉及这种语言非常有用的一些特性，但还远不能满足你的使用需求。因特网上有很多免费资源，如果偏爱读书（一定是吧，因为你正读着呢），可以看看《JavaScript 语言精粹》，相比《JavaScript 权威指南》而言，前者很薄，更易理解（后者也很棒，但是要多出 700 页）。这里不能全面介绍 JavaScript 的有用特性，但这门语言真的非常灵活且功能强大。

5.8 技巧46：在shell中管理所有服务器和数据库

默认情况下，*mongo* 会连接 *localhost:27017*。启动时可以任意指定要连接的服务器，方法是运行 *mongo* 主机 : 端口 / 数据库。在 shell 下还可以连接多个服务器或者数据库。

例如，假设应用需要两个数据库：一个 *customers* 数据库，一个 *game* 数据库。同时使用两者且需要在两者之间切换时可以用 `use customers`、`use game`、`use customers` 等。也可以用不同的变量表示不同的数据库：

```
> db
test
> customers = db.getSisterDB("customers")
customers
> game = db.getSisterDB("game")
game
```

这些变量和 `db` 的用法一样，如 `game.players.find()`、`customers.europe.update()` 等。

也可以将 `db` 或者别的变量指向其他服务器：

```
> db = connect("ny1a:27017/foo")
connecting to: ny1a:27017/foo
foo
```

```
> db
foo
```

若运行复制组或者分片集群时需要连接多个节点，这样的技巧就很方便了。在 shell 中可以同时维护到主节点和从属节点的连接：

```
> master = connect("ny1a:27017/admin")
connecting to: ny1a:27017/admin
admin
> slave = connect("ny1b:27017/admin")
connecting to: ny1b:27017/admin
admin
```

你还能直接连接分片服务器、配置服务器，只要是运行的 MongoDB 服务器就可以。

> 一些 shell 函数，尤其是 `rs` 辅助函数，是以 `db` 作为当前数据库的，如果 `db` 连接的是从属节点或者仲裁机，有些辅助函数就失效了。

利用 shell 连接多个服务器有个令人讨厌的地方，即 MongoDB 会跟踪发起的所有连接，一旦连接丢失，就不停地报警，直到连接恢复或者重启 shell。即便是清除连接也无效！这个错误会在版本 1.9 中得以修正，不过现在虽然有点恼人却也无关紧要。

5.9 技巧47：获得帮助

Java Script 使得我们可以在 shell 中看到绝大部分函数的源代码。要想知道函数接受什么参数或者记不住返回值，你就可以通过执行函数名但不加括号来查看源代码。例如，假设我们记得 `db.addUser` 用来添加一个用户，但拿不准参数是什么了：

```
> db.addUser
function (username, pass, readOnly) {
    readOnly = readOnly || false;
    var c = this.getCollection("system.users");
    var u = c.findOne({user: username}) || {user: username};
    u.readOnly = readOnly;
    u.pwd = hex_md5(username + (":mongo:" + pass));
    print(tojson(u));
    c.save(u);
}
```

我们立刻就能知道，应该传一个 `username`（用户名）、`pass`（密码），还有个只读的参数 `read Only`（创建给定数据库的只读用户）。

另外，大家也可以在线查看 JavaScript API（http：//api.mongodb.org/js）。在线文档虽然写得不是很好，但还算比较完整的函数参考。

还有很多内置的命令帮助。要是想不起来要执行的命令也没关系，只需记住一个命令——`listCommands`。它会将所有命令的名称都列出来。

```
> db.runCommand({listCommands : 1})
{
"commands" : {
"_isSelf" : { ... },
...
}
"ok" : 1
}
```

如果知道命令名，就可以通过 {*commandName* : 1, help : 1}（即便 1 对于这个命令本身没有意义也没关系）来查看内置文档。这种方式可以显示各个命令在数据据库中的基本文档，有些很有用，有的写得不怎么样。

```
> db.runCommand({collstats : 1, help : 1})
{
"help" : "help for: collStats { collStats: "blog.posts" , scale : 1 }
scale divides sizes e.g. for KB use 1024",
"lockType" : -1,
"ok" : 1
}
```

shell 还有 tab 补全功能，因此你可以获得函数名、字段，甚至是现有集合名的输入提示信息：

```
> db.c
db.cloneCollection(   db.constructor      db.currentOP(
db.cloneDatabase(     db.copyDatabase(    db.currentOp(
db.commandHelp(       db.createCollection(
> db.copyDatabase()
```

编写本书时，只有 *NIX 系统上实现了 shell 补全。

5.10 技巧48：创建启动文件

shell 启动时可以运行启动文件。启动文件通常含有一组用户自定义的辅助函数，但也无非就是个普通的 JavaScript 程序。要构建启动文件，建一个以 .js 为后缀的文件（比方说 startup.js）就可以了，然后用 `mongo startup.js` 启动。

比如，假如要做些 shell 维护工作，要避免意外删除数据库或者记录。你就可以在 shell 中删除一些不太安全的命令（比如删除数据库、集合、文档等）：

```
// no-delete.js
delete DBCollection.prototype.drop;
delete DBCollection.prototype.remove;
delete DB.prototype.dropDatabase;
```

这样，删除集合时，*mongo* 无法识别这一函数：

```
$ mongo no-delete.js
MongoDB shell version: 1.8.0
connecting to: test
> db.foo.drop()
Wed Feb 16 14:24:16 TypeError: db.foo.drop is not a function (shell):1
```

但这种招数“防君子不防小人”，对于那些下定决心删除集合的人一点儿作用都没有。所以删除函数并不能作为防范恶意攻击的手段（本身也不提供这一作用），它仅仅用于避免些误操作而已。

> 要是有人真的铁了心想删除集合，但找不到 `drop()`，只要执行 `db.$cmd.findOne( {drop : "foo"})` 就可以了。要是连这个都不让做就只能把 `find()` 也删掉，但 shell 也就没什么用处了。

你可以建立一个详尽的函数黑名单，这需要考虑实际情况（是要禁止创建索引，还是禁止执行数据库命令等）。*mongo* 启动时可以指定多个启动文件，所以可以实现模块化操作。

5.11 技巧49：自定义函数

如果你想创建自定义函数，可以把它们定义为全局函数，或者添加到类的实例，或类本身中（这样类的每个实例都会包含这个函数的一个实例）。

例如，假设像技巧 46 那样连接复制组中的所有节点，需要添加一个 `getOplogLength` 函数。

经过一番考虑，我们决定将其添加到数据库类（DB）中：

```
DB.prototype.getOplogLength = function() {
    var local = this.getSisterDB("local");
    var first = local.oplog.rs.find().sort({$natural : 1}).limit(1).next();
    var last = local.oplog.rs.find().sort({$natural : -1}).limit(1).next();
    print("total time: " + (last.ts.t - first.ts.t) + " secs");
};
```

然后，连接 *rsA*、*rsB* 和 *rsC* 这些数据库时就都会有 `getOplogLength` 函数。

要是事先已经在使用 *rsA*、*rsB* 和 *rsC*，则即使你添加了新函数到（它们用来进实例化的）类中它们也不能用。（JavaScript 中的类相当于类实例的模板，一旦初始化完成，实例和类就没有关系了）。如果连接已经初始化了，就得为每个实例逐个添加这一方法：

```
// 将函数存入一个变量以保存输入
var f = function() { ... }
rsA.getOplogSize = f;
rsB.getOplogSize = f;
rsC.getOplogSize = f;
```

你也可以稍作修改，将其作为全局函数：

```
getOplogLength = function(db) {
    var local = db.getSisterDB("local");
    ...
};
```

当然也可以在对象（以及对象的方法）上做类似操作。

从文件加载JavaScript

在 shell 中可以随时用 `load()` 函数加载 JavaScript 库。`load()` 加载 JavaScript 文件并在 shell 上下文中执行（因此 shell 中所有的全局变量对其可见）。你也可以在加载的文件中定义要在 shell 中全局使用的变量。你也可以在这些文件中用 `print` 函数将输出呈现在 shell 中：

```
// hello.js
print("Hello, world!")
```

然后在 shell 中执行：

```
> load("hello.js")
Hello, world!
```

操作人员通常想要用配置文件设置复制组或者分片。这个复制组和分片的设置过程必须通过编程完成，但你可以把设置函数写入 JavaScript 文件，然后执行它就可以建立复制组。这和使用配置文件也差不多了。

5.12　技巧50：使用单个连接读取自身写入

MongoDB 服务器的连接就像一个请求队列。比方说，若通过这个连接向数据库依次发送请求 A、B、C，则 MongoDB 会按照 A、B、C 的发送顺序处理。但并不保证每个操作都能成功执行，可能 A 就关停了服务器（为 `shutdownServer` 命令），然后 B 和 C 都会返回错误（要求返回执行结果的情况下）。但是，其发送和执行的顺序是绝对能得到保证的（见图 5-1）。

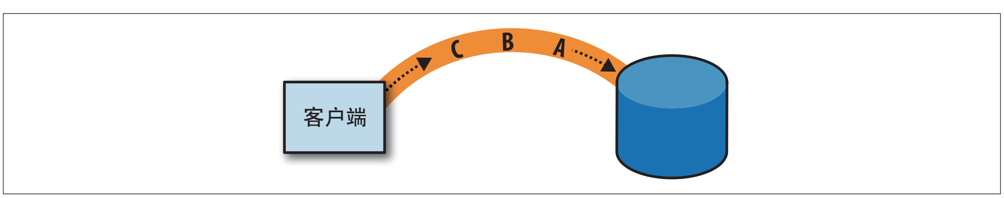


图 5-1：到 MongoDB 的单个连接类似队列

这是很有用的。例如假设增加了某个产品的下载次数，然后用 `findOne` 查询，你会希望得到增加后的下载次数。但是，要是用的是多个连接（多数驱动都会自动使用连接池），可能就会事与愿违。

假设有两个到数据库的连接（同一个客户端发起的）。每个连接的请求都能按顺序处理，但两个连接之间就没有什么确定的先后顺序了。如果第一个连接发送了 A、B、C 这 3 个请求，第二个连接发送了 D、E、F 这 3 个请求，最后处理顺序可能是 A、D、B、E、C、F，也可能是 A、B、C、D、E、F，也可能是两个序列的其他组合（见图 5-2）

如果 A 请求是插入新文档，D 请求是查询这个文档，D 最后可能跑在前面了（比如 D、A、E、B、F、C 这样的顺序），这样就找不到记录了。为了修正这一点，有连接池的驱动一般都会提供一个方法，让一组请求通过同一个连接发送，来避免这种“读取自己的写入”的矛盾。其他的驱动会自动这样处理（通常每个“会话”使用同一个连接），详见驱动文档。

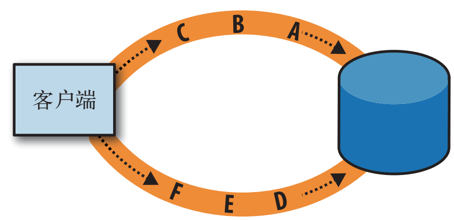


图 5-2：请求在同一个连接内能保证处理顺序，但是在不同的连接中就无法保证了

深入学习MongoDB

本书分两部分，分别来自O'Reilly的《MongoDB扩展技术》与《MongoDB开发技巧50例》两书。

前一部分“MongoDB扩展技术”指导大家创建一个不断增长以满足应用程序需求的MongoDB集群，内容简明扼要，指导用户设置和使用集群存储大量数据并高效访问数据。此外，读者还可了解如何让应用程序兼容分布式数据库系统。遵照其中建议，你很快就可通过MongoDB构建和运行一个高效的、可预测的分布式系统。

具体的主题有：

- 通过分片设置MongoDB集群；
- 在集群中查询和更新数据；
- 操作、监控和备份集群；
- 从程序设计角度，考虑如何应对分片、配置服务器或者*mongos*进程停止运行的情况。

对于用户而言，MongoDB上手很容易，但是构建使用MongoDB的应用程序时，一些棘手的问题便会接踵而来。怎样权衡范式化与反范式化？怎样处理复制组失效的情况并进行故障恢复？本书第二部分“MongoDB开发技巧50例”呈现了一系列的MongoDB提示和技巧，可帮助用户解决与应用程序设计与实现、数据安全和监控有关的各种问题。

内容涵盖10gen公司工程师的实际指导，并通过以下5个话题展开了论述。

- 应用设计技巧：模式设计阶段应注意的问题
- 实现技巧：基于MongoDB编写应用程序
- 优化技巧：为应用提速
- 数据安全技巧：在不牺牲太多性能的情况下，利用复制和日志保证数据安全
- 管理技巧：配置MongoDB并确保其平滑运行

Kristina Chodorow 10gen公司的软件工程师，MongoDB项目的核心成员，从事与数据库服务器、PHP驱动、Perl驱动等相关的工作。她常在世界级技术大会上作报告，包括OSCON、LinuxCon、FOSDEM和Latinoware。

封面设计：Karen Montgomery 张健

图灵社区：www.ituring.com.cn
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

O'REILLY®
oreilly.com.cn

分类建议 计算机 / 数据库 / MongoDB

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-27211-9

9 787115 272119 >

ISBN 978-7-115-27211-9
定价：32.00元

Sharding, Cluster Setup, and Administration

Scaling MongoDB



O'REILLY®

Kristina Chodorow

Scaling MongoDB

Scaling MongoDB

Kristina Chodorow

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Scaling MongoDB
by Kristina Chodorow

Copyright © 2011 Kristina Chodorow. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Production Editor: Holly Bauer
Proofreader: Holly Bauer
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
February 2011: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Scaling MongoDB*, the image of a trigger fish, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-30321-1

[LSI]

1296240830

Table of Contents

Preface

This text is for MongoDB users who are interested in sharding. It is a comprehensive look at how to set up and use a cluster.

This is *not* an introduction to MongoDB; I assume that you understand what a document, collection, and database are, how to read and write data, what an index is, and how and why to set up a replica set.

If you are not familiar with MongoDB, it's easy to learn. There are a number of books on MongoDB, including *MongoDB: The Definitive Guide* from this author. You can also check out the online documentation.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Scaling MongoDB* by Kristina Chodorow (O'Reilly). Copyright 2011 Kristina Chodorow, 978-1-449-30321-1."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North

Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/9781449303211

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1

Welcome to Distributed Computing!

In the *Terminator* movies, an artificial intelligence called Skynet wages war on humans, chugging along for decades creating robots and killing off humanity. This is the dream of most ops people—not to destroy humanity, but to build a distributed system that will work long-term without relying on people carrying pagers. Skynet is still a pipe dream, unfortunately, because distributed systems are very difficult, both to design well and to keep running.

A single database server has a couple of basic states: it's either up or down. If you add another machine and divide your data between the two, you now have some sort of dependency between the servers. How does it affect one machine if the other goes down? Can your application handle either (or both) machines going down? What if the two machines are up, but can't communicate? What if they can communicate, but only very, very, slowly?

As you add more nodes, these problems just become more numerous and complex: what happens if entire parts of your cluster can't communicate with other parts? What happens if one subset of machines crashes? What happens if you lose an entire data center? Suddenly, even taking a backup becomes difficult: how do you take a consistent snapshot of many terabytes of data across dozens of machines without freezing out the application trying to use the data?

If you can get away with a single server, it is much simpler. However, if you want to store a large volume of data or access it at a rate higher than a single server can handle, you'll need to set up a cluster. On the plus side, MongoDB tries to take care of a lot of the issues listed above. Keep in mind that this isn't as simple as setting up a single *mongod* (then again, what is?). This book shows you how to set up a robust cluster and what to expect every step of the way.

What Is Sharding?

Sharding is the method MongoDB uses to split a large collection across several servers (called a *cluster*). While sharding has roots in relational database partitioning, it is (like most aspects of MongoDB) very different.

The biggest difference between any partitioning schemes you've probably used and MongoDB is that MongoDB does almost everything automatically. Once you tell MongoDB to distribute data, it will take care of keeping your data balanced between servers. You have to tell MongoDB to add new servers to the cluster, but once you do, MongoDB takes care of making sure that they get an even amount of the data, too.

Sharding is designed to fulfill three simple goals:

Make the cluster "invisible."
: We want an application to have no idea that what it's talking to is anything other than a single, vanilla *mongod*.

  To accomplish this, MongoDB comes with a special routing process called *mongos*. *mongos* sits in front of your cluster and looks like an ordinary *mongod* server to anything that connects to it. It forwards requests to the correct server or servers in the cluster, then assembles their responses and sends them back to the client. This makes it so that, in general, a client does not need to know that they're talking to a cluster rather than a single server.

  There are a couple of exceptions to this abstraction when the nature of a cluster forces it. These are covered in Chapter 4.

Make the cluster always available for reads and writes.
: A cluster can't guarantee it'll always be available (what if the power goes out everywhere?), but within reasonable parameters, there should never be a time when users can't read or write data. The cluster should allow as many nodes as possible to fail before its functionality noticeably degrades.

  MongoDB ensures maximum uptime in a couple different ways. Every part of a cluster can and should have at least some redundant processes running on other machines (optimally in other data centers) so that if one process/machine/data center goes down, the other ones can immediately (and automatically) pick up the slack and keep going.

  There is also the question of what to do when data is being migrated from one machine to another, which is actually a very interesting and difficult problem: how do you provide continuous and consistent access to data while it's in transit? We've come up with some clever solutions to this, but it's a bit beyond the scope of this book. However, under the covers, MongoDB is doing some pretty nifty tricks.

Let the cluster grow easily
: As your system needs more space or resources, you should be able to add them.

MongoDB allows you to add as much capacity as you need as you need it. Adding (and removing) capacity is covered further in Chapter 3.

These goals have some consequences: a cluster should be easy to use (as easy to use as a single node) and easy to administrate (otherwise adding a new shard would not be easy). MongoDB lets your application grow—easily, robustly, and naturally—as far as it needs to.

CHAPTER 2
Understanding Sharding

To set up, administrate, or debug a cluster, you have to understand the basic scheme of how sharding works. This chapter covers the basics so that you can reason about what's going on.

Splitting Up Data

A *shard* is one or more servers in a cluster that are responsible for some subset of the data. For instance, if we had a cluster that contained 1,000,000 documents representing a website's users, one shard might contain information about 200,000 of the users.

A shard can consist of many servers. If there is more than one server in a shard, each server has an identical copy of the subset of data (Figure 2-1). In production, a shard will usually be a replica set.

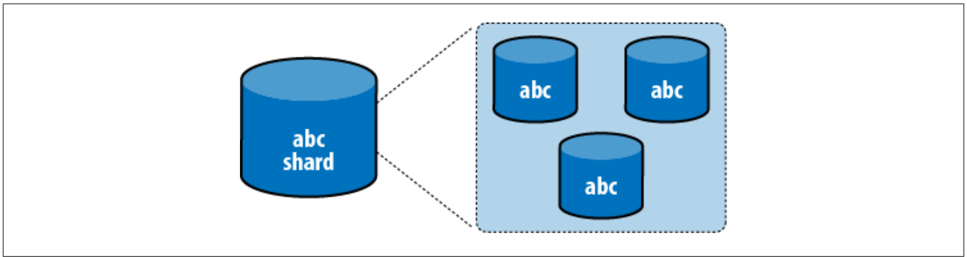


Figure 2-1. A shard contains some subset of the data. If a shard contains more than one server, each server has a complete copy of the data.

To evenly distribute data across shards, MongoDB moves subsets of the data from shard to shard. It figures out which subsets to move based on a key that you choose. For example, we might choose to split up a collection of users based on the *username* field. MongoDB uses range-based splitting; that is, data is split into chunks of given ranges —e.g., *["a", "f")*.

Throughout this text, I'll use standard range notation to describe ranges. "*[*" and "*]*" denote inclusive bounds and "*(*" and "*)*" denote exclusive bounds. Thus, the four possible ranges are:

x is in (a, b)
: If there exists an x such that $a < x < b$

x is in (a, b]
: If there exists an x such that $a < x \leq b$

x is in [a, b)
: If there exists an x such that $a \leq x < b$

x is in [a, b]
: If there exists an x such that $a \leq x \leq b$

MongoDB's sharding uses *[a, b)* for almost all of its ranges, so that's mostly what you'll see. This range can be expressed as "from and including a, up to but not including b."

For example, say we have a range of username *["a", "f")*. Then "a", "charlie", and "ez-bake" could be in the set, because, using string comparison, "a" ≤ "a" < "charlie" < "ez-bake" < "f".

The range includes everything *up to but not including* "f". Thus, "ez-bake" could be in the set, but "f" could not.

Distributing Data

MongoDB uses a somewhat non-intuitive method of partitioning data. To understand why it does this, we'll start by using the naïve method and figure out a better way from the problems we run into.

One range per shard

The simplest way to distribute data across shards is for each shard to be responsible for a single range of data. So, if we had four shards, we might have a setup like Figure 2-2. In this example, we will assume that all usernames start with a letter between "a" and "z", which can be represented as *["a", "{")*. "*{*" is the character after "z" in ASCII.

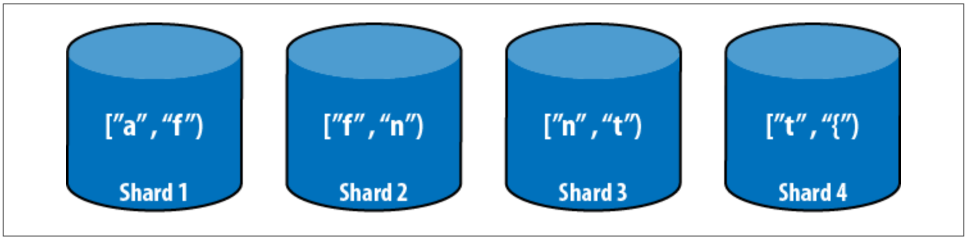


Figure 2-2. Four shards with ranges ["a", "f"), ["f", "n"), ["n", "t"), and ["t", "{")

This is a nice, easy-to-understand system for sharding, but it becomes inconvenient in a large or busy system. It's easiest to see why by working through what would happen.

Suppose a lot of users start registering names starting with *["a", "f")*. This will make Shard 1 larger, so we'll take some of its documents and move them to Shard 2. We can adjust the ranges so that Shard 1 is (say) *["a", "c")* and Shard 2 is *["c", "n")* (see Figure 2-3).

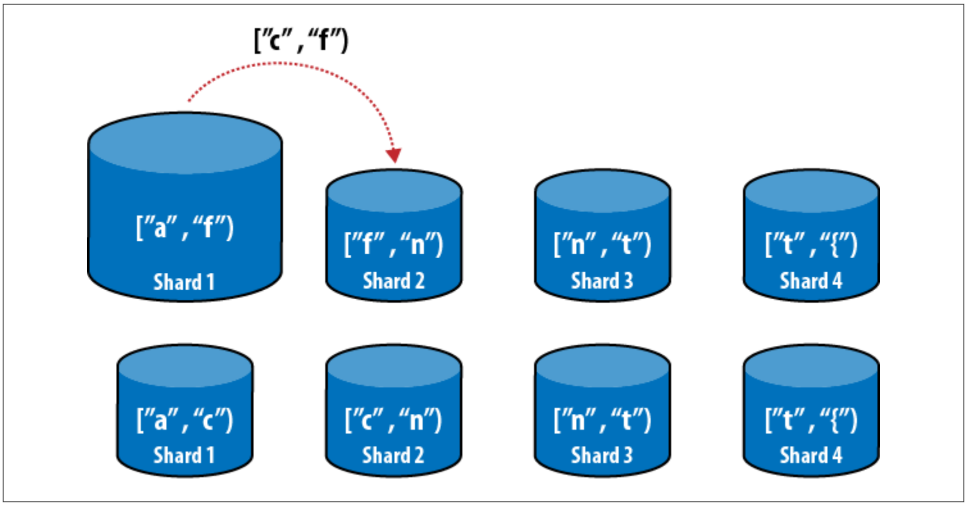


Figure 2-3. Migrating some of Shard 1's data to Shard 2. Shard 1's range is reduced and Shard 2's is expanded.

Everything seems okay so far, but what if Shard 2 is getting overloaded, too? Suppose Shard 1 and Shard 2 have 500GB of data each and Shard 3 and Shard 4 only have 300GB each. Given this sharding scheme, we end up with a cascade of copies: we'd have to move 100GB from shard 1 to Shard 2, then 200GB from shard 2 to shard 3, then 100GB from shard 3 to shard 4, for a total of 400GB moved (Figure 2-4). That's a lot of extra data moved considering that all movement has to cascade across the cluster.

How about adding a new shard? Let's say this cluster keeps working and eventually we end up having 500GB per shard and we add a new shard. Now we have to move 400GB from Shard 4 to Shard 5, 300GB from Shard 3 to Shard 4, 200GB from Shard 2 to Shard 3, 100GB from Shard 1 to Shard 2 (Figure 2-5). That's 1TB of data moved!

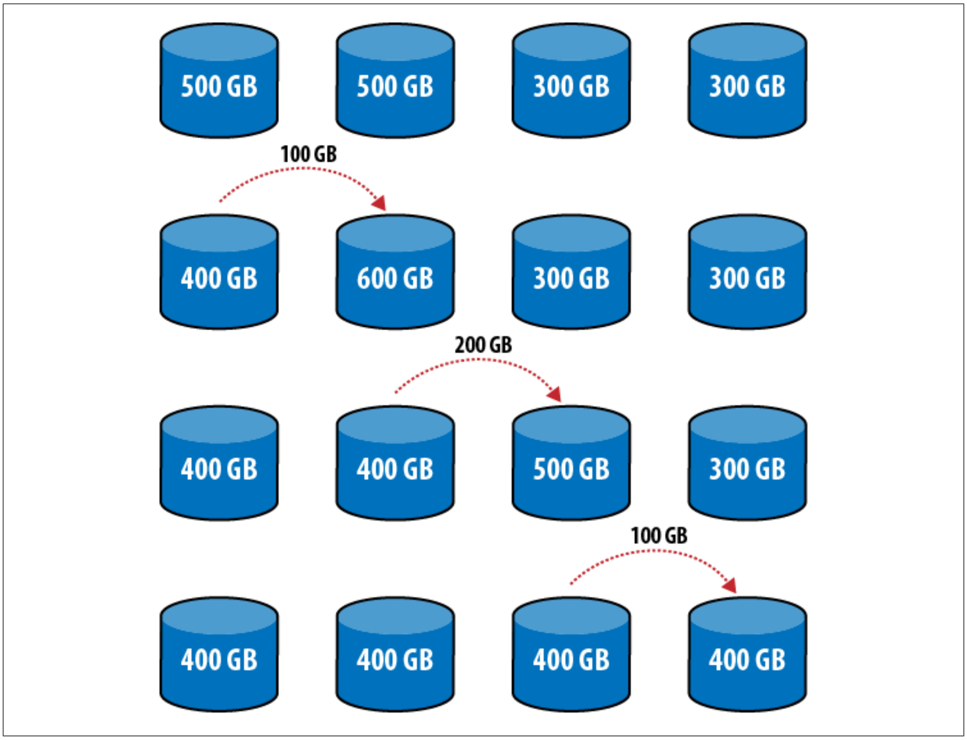


Figure 2-4. Using a single range per shard creates a cascade effect: data has to be moved to the server "next to" it, even if that does not improve the balance

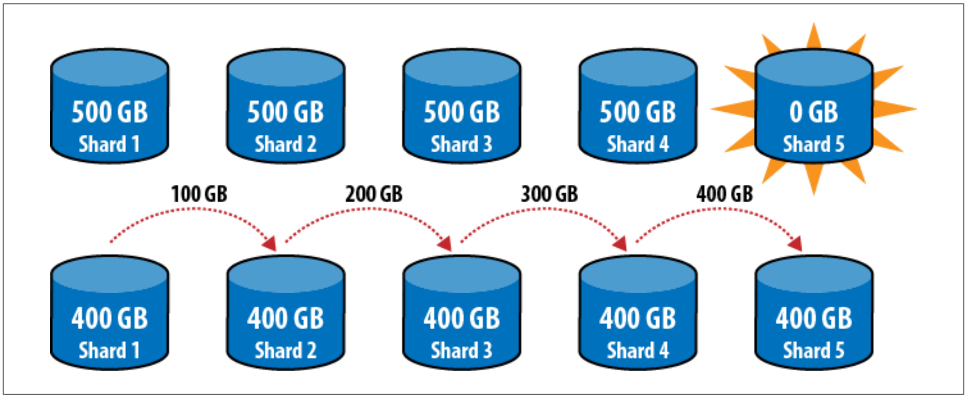


Figure 2-5. Adding a new server and balancing the cluster. We could cut down on the amount of data transferred by adding the new server to the "middle" (between Shard 2 and Shard 3), but it would still require 600GB of data transfer.

This cascade situation just gets worse and worse as the number of shards and amount of data grows. Thus, MongoDB *does not* distribute data this way. Instead, each shard contains multiple ranges.

Multi-range shards

Let's consider the situation pictured in Figure 2-4 again, where Shard 1 and Shard 2 have 500GB and Shard 3 and Shard 4 have 300GB. This time, we'll allow each shard to contain multiple chunk ranges.

This allows us to divide Shard 1's data into two ranges: one of 400GB (say *["a", "d")*) and one of 100GB (*["d", "f")*). Then, we'll do the same on Shard 2, ending up with *["f", "j")* and *["j", "n")*. Now, we can migrate 100GB (*["d", "f")*) from Shard 1 to Shard 3 and all of the documents in the *["j", "n")* range from Shard 2 to Shard 4 (see Figure 2-6). A range of data is called a *chunk*. When we split a chunk's range into two ranges, it becomes two chunks.

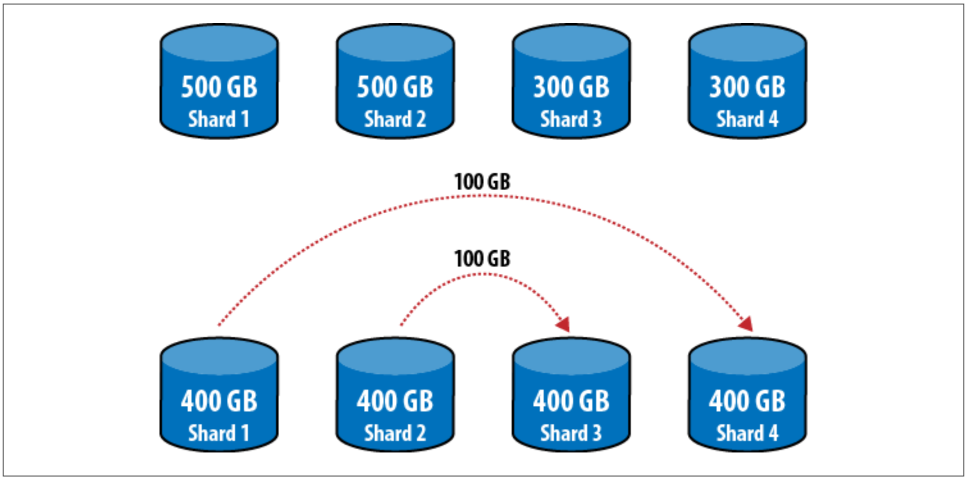


Figure 2-6. Allowing multiple, non-consecutive ranges in a shard allows us to pick and choose data and to move it anywhere

Now there are 400GB of data on each shard and only 200GB of data had to be moved.

If we add a new shard, MongoDB can skim 100GB off of the top of each shard and move these chunks to the new shard, allowing the new shard to get 400GB of data by moving the bare minimum: only 400GB of data (Figure 2-7).

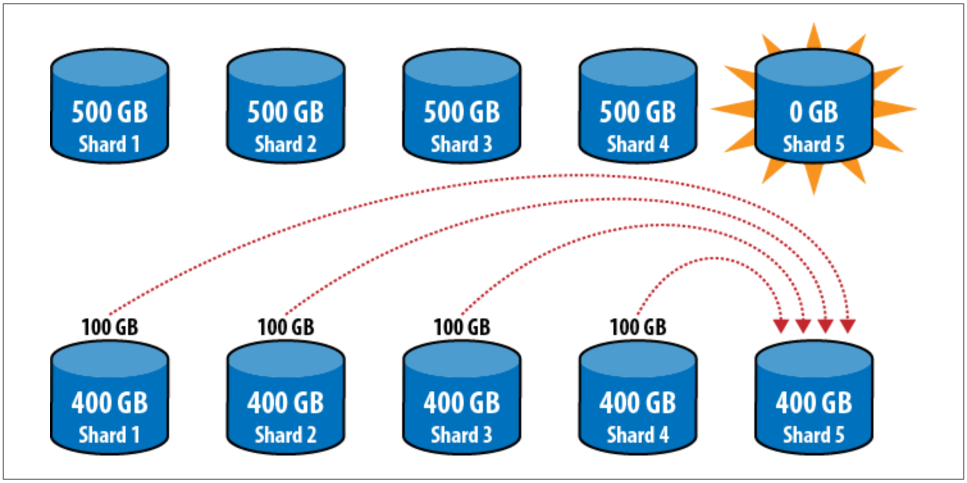


Figure 2-7. When a new shard is added, everyone can contribute data to it directly

This is how MongoDB distributes data between shards. As a chunk gets bigger, MongoDB will automatically split it into two smaller chunks. If the shards become unbalanced, chunks will be migrated to correct the imbalance.

How Chunks Are Created

When you decide to distribute data, you have to choose a key to use for chunk ranges (we've been using `username` above). This key is called a *shard key* and can be any field or combination of fields. (We'll go over how to choose the shard key and the actual commands to shard a collection in Chapter 3.)

Example

Suppose our collection had documents that looked like this (*_ids* omitted):

```
{"username" : "paul", "age" : 23}
{"username" : "simon", "age" : 17}
{"username" : "widdly", "age" : 16}
{"username" : "scuds", "age" : 95}
{"username" : "grill", "age" : 18}
{"username" : "flavored", "age" : 55}
{"username" : "bertango", "age" : 73}
{"username" : "wooster", "age" : 33}
```

If we choose the *age* field as a shard key and end up with a chunk range *[15, 26)*, the chunk would contain the following documents:

```
{"username" : "paul", "age" : 23}
{"username" : "simon", "age" : 17}
{"username" : "widdly", "age" : 16}
{"username" : "grill", "age" : 18}
```

As you can see, all of the documents in this chunk have their *age* value in the chunk's range.

Sharding collections

When you first shard a collection, MongoDB creates a single chunk for whatever data is in the collection. This chunk has a range of *(-∞, ∞)*, where -∞ is the smallest value MongoDB can represent (also called `$minKey`) and ∞ is the largest (also called `$maxKey`).

> If you shard a collection containing a lot of data, MongoDB will immediately split this initial chunk into smaller chunks.

The collection in the example above is too small to actually trigger a split, so you'd end up with a single chunk—*(-∞, ∞)*—until you inserted more data. However, for the purposes of demonstration, let's pretend that this was enough data.

MongoDB would split the initial chunk *(-∞, ∞)* into two chunks around the midpoint of the existing data's range. So, if approximately half of the documents had a an *age* field less than 15 and half were greater than 15, MongoDB might choose 15. Then we'd end up with two chunks: *(-∞, 15)*, *[15, ∞)* (Figure 2-8). If we continued to insert data into the *[15, ∞)* chunk, it could be split again, into, say, *[15, 26)* and *[26, ∞)*. So now we have three chunks in this collection: *(-∞, 15)*, *[15, 26)*, and *[26, ∞)*. As we insert more data, MongoDB will continue to split existing chunks to create new ones.

You can have a chunk with a single value as its range (e.g., only users with the username "paul"), but every chunk's range must be distinct (you cannot have two chunks with the range *["a", "f")*). You also cannot have overlapping chunks; each chunk's range must exactly meet the next chunk's range. So, if you split a chunk with the range *[4, 8)*, you could end up with *[4, 6)* and *[6, 8)* because together, they fully cover the original chunk's range. You could not have *[4, 5)* and *[6, 8)* because then your collection is missing everything in *[5, 6)*. You could not have *[4, 6)* and *[5, 8)* because then chunks would overlap. Each document must belong to one and only one chunk.

As MongoDB does not enforce any sort of schema, you might be wondering: where is a document placed if it doesn't have a value for the shard key? MongoDB won't actually allow you to insert documents that are missing the shard key (although using `null` for the value is fine). You also cannot change the value of a shard key (with, for example, a `$set`). The only way to give a document a new shard key is to remove the document, change the shard key's value on the client side, and reinsert it.

What if you use strings for some documents and numbers for others? It works fine, as there is a strict ordering between types in MongoDB. If you insert a string (or an array, boolean, `null`, etc.) in the *age* field, MongoDB would sort it according to its type. The ordering of types is:

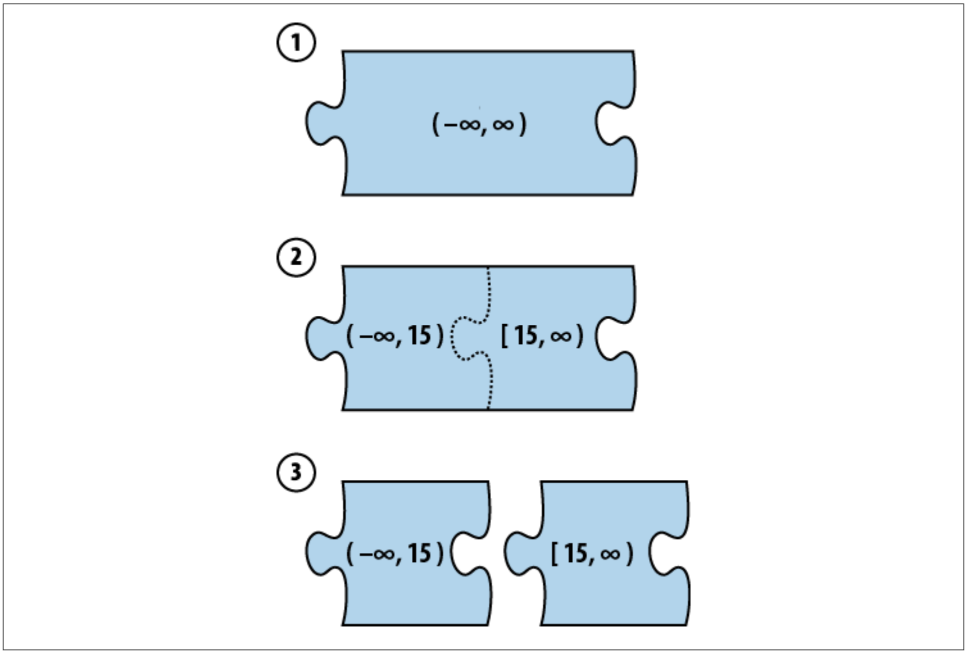


Figure 2-8. A chunk splitting into two chunks

> null < numbers < strings < objects < arrays < binary data < ObjectIds < booleans < dates < regular expressions

Within a type, orderings are as you'd probably expect: 2 < 4, "a" < "z".

In the first example given, chunks are hundreds of gigabytes in size, but in a real system, chunks are only 200MB by default. This is because moving data is expensive: it takes a lot of time, uses system resources, and can add a significant amount of network traffic. You can try it out by inserting 200MB into a collection. Then try fetching all 200MB of data. Then imagine doing this on a system with multiple indexes (as your production system will probably have) while other traffic is coming in. You don't want your application to grind to a halt while MongoDB shuffles data in the background; in fact, if a chunk gets too big, MongoDB will refuse to move it at all. You don't want chunks to be too small, either, because each chunk has a little bit of administrative overhead to requests (so you don't want to have to keep track of zillions of them). It turns out that 200MB is the sweet spot between portability and minimal overhead.

> A chunk is a logical concept, not a physical reality. The documents in a chunk are not physically contiguous on disk or grouped in any way. They may be scattered at random throughout a collection. A document belongs in a chunk if and only if its shard key value is in that chunk's range.

Balancing

If there are multiple shards available, MongoDB will start migrating data to other shards once you have a sufficient number of chunks. This migration is called *balancing* and is performed by a process called the *balancer*.

The balancer moves chunks from one shard to another. The nice thing about the balancer is that it's automatic—you don't have to worry about keeping your data even across shards because it's done for you. This is also the downside: it's automatic, so if you don't like the way it's balancing things, tough luck. If you decide you don't want a certain chunk on Shard 3, you can manually move it to Shard 2, but the balancer will probably just pick it up and move it back to Shard 3. Your only options are to either re-shard the collection or turn off balancing.

As of this writing, the balancer's algorithm isn't terribly intelligent. It moves chunks based on the overall size of the shard and calls it a day. It will become more advanced in the (near) future.

The goal of the balancer is not only to keep the data evenly distributed but also to minimize the amount of data transferred. Thus, it takes a lot to trigger the balancer. For a balancing round to occur, a shard must have at least nine more chunks than the least-populous shard. At that point, chunks will be migrated off of the crowded shard until it is even with the rest of the shards.

The reason the balancer isn't very aggressive is that MongoDB wants to avoid sending the same data back and forth. If the balancer balanced out any tiny difference, it could constantly waste resources: Shard 1 would have two chunks more than Shard 2, so it would send Shard 2 one chunk. Then a few writes would go to Shard 2, and Shard 2 would end up with two more chunks than Shard 1 and send the original chunk right back (Figure 2-9). By waiting for a more severe imbalance, MongoDB can minimize pointless data transfers. Keep in mind that nine chunks is not even that much of an imbalance—it is less than 2GB of data.

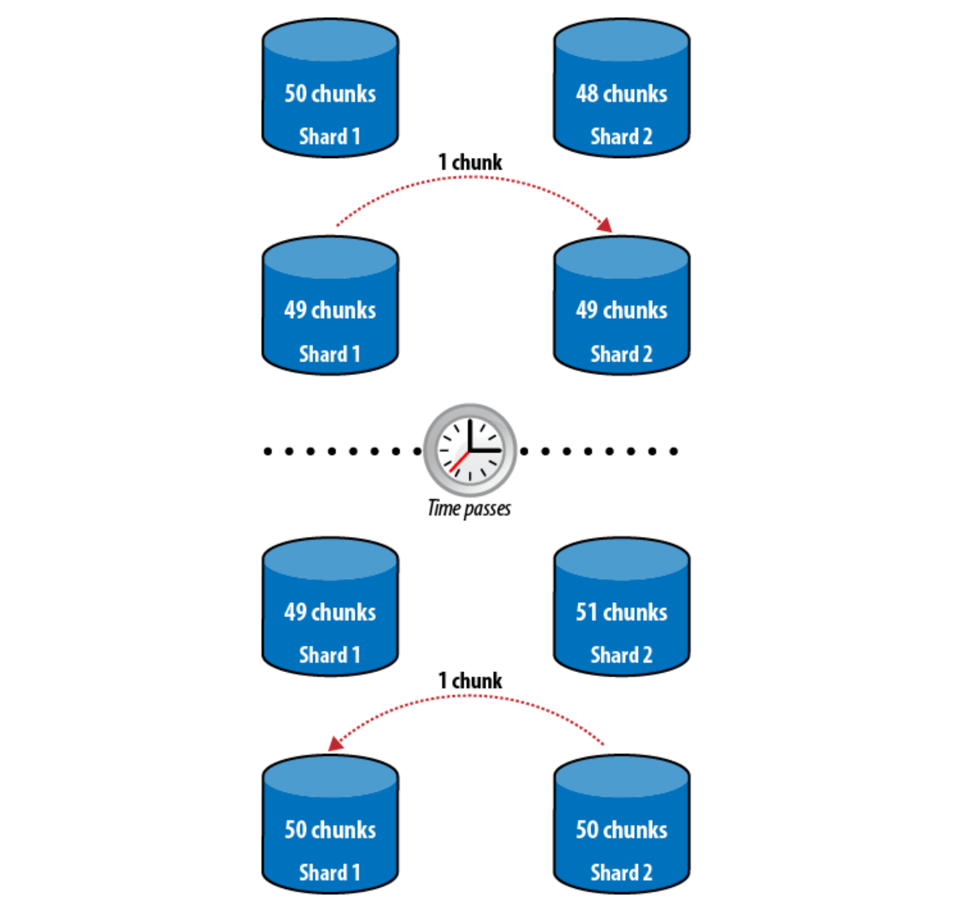


Figure 2-9. If every slight imbalance is corrected, a lot of data will end up moving unnecessarily

The Psychopathology of Everyday Balancing

Most users want to prove to themselves that sharding works by watching their data move, which creates a problem: the amount of data it takes to trigger a balancing round is larger than most people realize.

Let's say I'm just playing with sharding, so I write a shell script to insert half a million documents into a sharded collection.

```
> for (i=0; i<500000; i++) {
    db.foo.insert({"_id" : i, "x" : 1,"y" : 2, "z" : i, "date" : new Date(),
      "foo" : "bar"});
}
```

After half a million documents, I should see some data flying around, right? Wrong. If I take a look at the database stats, I still have a ways to go (some fields have been omitted for clarity):

```
> db.stats()
{
    "raw" : {
        "ubuntu:27017" : {
            /* shard stats */
        },
        "ubuntu:27018" : {
            /* shard stats */
        }
    },
    "objects" : 500008,
    "avgObjSize" : 79.99937600998383,
    "dataSize" : 40000328,
    "storageSize" : 69082624,
    "ok" : 1
}
```

If you look at *dataSize*, you can see that I have 40,000,328 bytes of data, which is roughly equivalent to 40MB. That's not even a chunk. That's not even a quarter of a chunk! To actually see data move, I would need to insert 2GB, which is 25 million of these documents, or 50 times as much data as I am currently inserting.

When people start sharding, they want to see their data moving around. It's human nature. However, in a production system, you don't want a lot of migration because it's a very expensive operation. So, on the one hand, we have the very human desire to see migrations actually happen. On the other hand, we have the fact that sharding won't work very well if it isn't irritatingly slow to human eyes.

What MongoDB did was implement two "tricks" to make sharding more satisfying for users, but not destructive to production systems.

Trick 1: Custom chunk size
: When you start up your first *mongos*, you can specify `--chunkSize N`, where `N` is the chunk size that you want in megabytes. If you're just trying out sharding and messing around, you can set `--chunkSize 1` and see your data migrating after a couple of megabytes of inserts.

Trick 2: Ramping up chunk size
: Even if you're deploying a real application, getting to 2GB can seem to take forever. So, for the first dozen or so chunks, MongoDB artificially lowers the chunk size automatically, from 200MB to 64MB, just to give people a more satisfying experience. Once you have a bunch of chunks, it ramps the chunk size back up to 200MB automatically.

On changing chunk size

Chunk size can be changed by starting with `--chunkSize N` or modifying the *config.settings* collection and restarting everything. However, unless you're trying out the 1MB chunk size for fun, *don't mess with the chunk size*.

I guarantee that, whatever you're trying to fix, fiddling with chunk size is not going to solve the root problem. It's a tempting knob to play with, but don't. Leave chunk size alone unless you want to play around with `--chunkSize 1`.

mongos

mongos is the interaction point between users and the cluster. Its job is to hide all of the gooey internals of sharding and present a clean, single-server interface to the user (you). There are a few cracks in this veneer, which are discussed in Chapter 4, but *mongos* mostly lets you treat a cluster as a single server.

When you use a cluster, you connect to a *mongos* and issue all reads and writes to that *mongos*. You should never have to access the shards directly (although you can if you want).

mongos forwards all user requests to the appropriate shards. If a user inserts a document, *mongos* looks at the document's shard key, looks at the chunks, and sends the document to the shard holding the correct chunk.

For example, say we insert `{"foo" : "bar"}` and we're sharding on `foo`. *mongos* looks at the chunks available and sees that there is a chunk with the range *["a", "c")*, which is the chunk that should contain "bar". This chunk lives on Shard 2, so *mongos* sends the insert message to Shard 2 (see Figure 2-10).

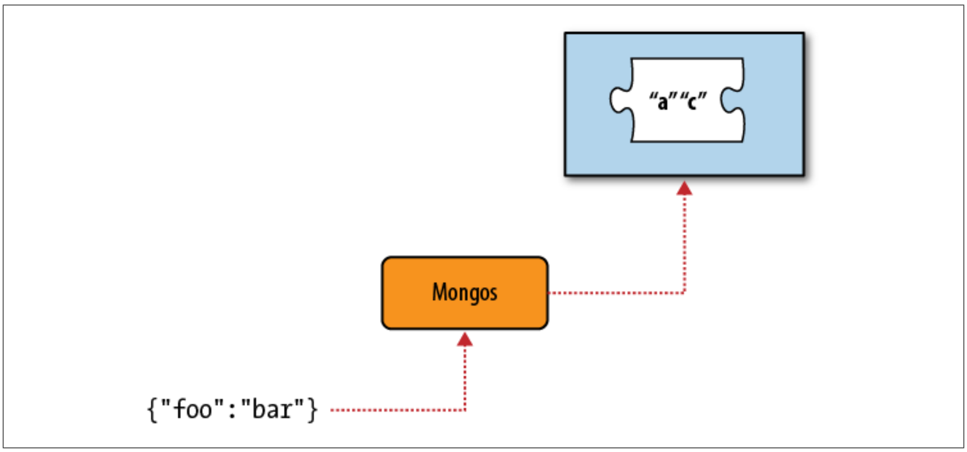


Figure 2-10. Routing a request to a shard with the corresponding chunk

If a query involves the shard key, *mongos* can use the same process it did to do the insert and find the correct shard (or shards) to send the query to. This is called a *targeted query* because it only targets shards that may have the data we're looking for. If it knows we're looking for `{"foo" : "bar"}`, there's no sense in querying a shard that only contains shard key values greater than "bar".

If the query does not contain the shard key, *mongos* must send the query to all of the shards. These can be less efficient than targeted queries, but isn't necessarily. A "spewed" query that accesses a few indexed documents in RAM will perform much better than a targeted query that has to access data from disk across many shards (a targeted query could hit every shard, too).

The Config Servers

mongos processes don't actually store any data persistently, so the configuration of a cluster is held on special *mongods* called *config servers*. Config servers hold the definitive information about the cluster for everyone's access (shards, *mongos* processes, and system administrators).

For a chunk migration to succeed, all of the configuration servers have to be up. If one of them goes down, all currently occurring migrations will revert themselves and stop until you get a full set of config servers up again. If any config servers go down, your cluster's configuration cannot change.

The Anatomy of a Cluster

A MongoDB cluster basically consists of three types of processes: the shards for actually storing data, the mongos processes for routing requests to the correct data, and the config servers, for keeping track of the cluster's state (Figure 2-11 and Figure 2-12).

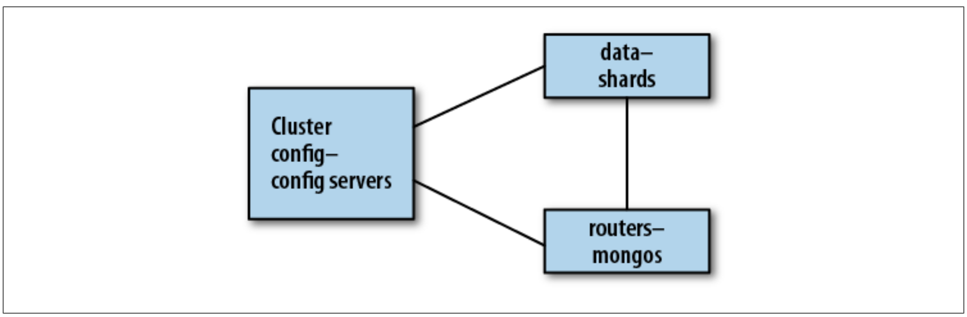


Figure 2-11. Three components of a cluster.

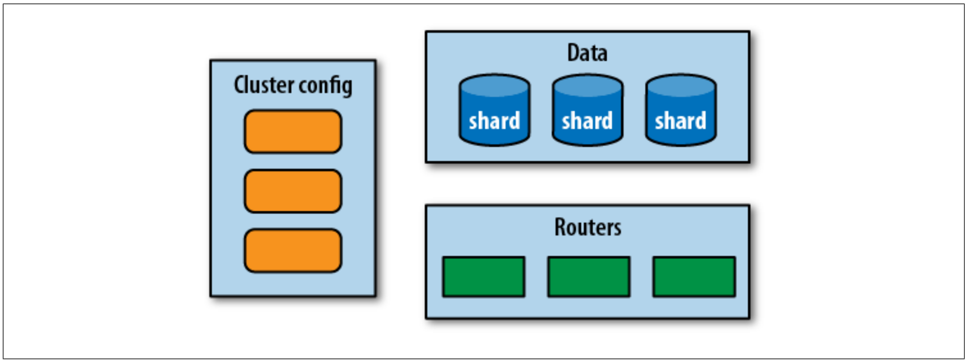


Figure 2-12. Each component can contain multiple processes.

Each of the components above is not "a machine." *mongos* processes are usually run on appservers. Config servers, for all their importance, are pretty lightweight and can basically be run on any machine available. Each shard usually consists of multiple machines, as that's where the data actually lives.

CHAPTER 3
Setting Up a Cluster

Choosing a Shard Key

Choosing a good shard key is absolutely critical. If you choose a bad shard key, it can break your application immediately or when you have heavy traffic, or it can lurk in wait and break your application at a random time.

On the other hand, if you choose a good shard key, MongoDB will just do the right thing as you get more traffic and add more servers, for as long as your application is up.

As you learned in the last chapter, a shard key determines how your data will be distributed across your cluster. Thus, you want a shard key that distributes reads and writes, but that also keeps the data you're using together. These can seem like contradictory goals, but it can often be accomplished.

First we'll go over a couple of bad shard key choices and find out why they're bad, then we'll come up with a couple of better ones. There is also a good page on the MongoDB wiki on choosing a shard key.

Low-Cardinality Shard Key

Some people don't really trust or understand how MongoDB automatically distributes data, so they think something along the lines of, "I have four shards, so I will use a field with four possible values for my shard key." This is a really, really bad idea.

Let's look at what happens.

Suppose we have an application that stores user information. Each document has a *continent* field, which is where the user is located. Its value can be "Africa", "Antarctica", "Asia", "Australia", "Europe", "North America", or "South America". We decide to shard on this key because we have a data center on each continent (okay, maybe not Antarctica) and we want to server people's data from their "local" data center.

The collection starts off with one chunk—*(-∞, ∞)*—on a shard in some data center. All your inserts and reads go to that one chunk. Once that chunk gets big enough, it'll

split into two chunks with ranges *(-∞, "Europe")* and *["Europe", ∞)*. All of the documents from Africa, Antarctica, Asia, or Australia go into the first chunk and all of the documents from Europe, North America, or South America will go into the second chunk. As you add more documents, eventually you'll end up with seven chunks (Figure 3-1):

- *(-∞, "Antarctica")*
- *["Antarctica", "Asia")*
- *["Asia", "Australia")*
- *["Australia", "Europe")*
- *["Europe", "North America")*
- *["North America", "South America")*
- *["South America", ∞)*

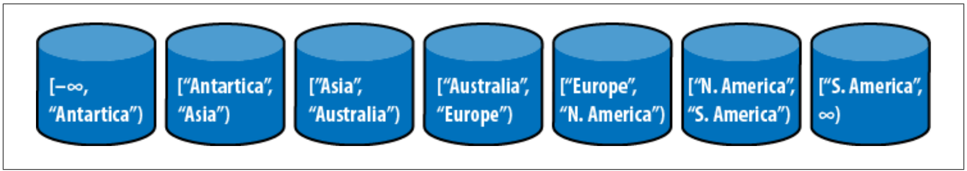


Figure 3-1. One shard per continent

Now what?

MongoDB can't split these chunks any further! The chunks will just keep getting bigger and bigger. This is fine for a while, but what happens when you start running out of space on your servers? There's nothing you can do, aside from getting a bigger hard disk.

This choice of shard key is called a *low-cardinality shard key*, because there are a limited number of shard key values. If you choose a shard key with low cardinality, you will end up with large, unmovable, unsplittable chunks that will make your life miserable.

If you are doing this because you want to manually distribute your data, do not use MongoDB's built-in sharding. You'll be fighting it all the way. However, you can certainly manually shard your collections, write your own router, and route reads and writes to whichever server(s) you'd like. It's just easier to choose a good shard key and let MongoDB do it for you.

Keys that this rule applies to

This rule applies to any key that has a finite number of values. Keep in mind that, if a key has `N` values in a collection, you can only have `N` chunks and, therefore, `N` shards.

If you are tempted to use low-cardinality shard key because you query on that field a lot, use a compound shard key (a shard key with two fields) and make sure that the second field has lots of different values MongoDB can use for splitting.

Exceptions to the rule

If a collection has a lifetime (e.g., you're creating a new collection each week and you know that, in a single week, you won't get near the capacity of any of your shards), you could choose this as a key.

Data center awareness

This example is not just about choosing a low-cardinality shard key, but also about trying to hack data-center awareness into MongoDB's sharding. Sharding does not yet support data center awareness. If you're interested in this, you can watch/vote for the relevant bug.

The problem with hacking it together yourself is that it isn't very extensible. What happens if your application is big in Japan? Now you want to add a second shard to handle Asia.

How are you going to migrate data over? You can't move a chunk once it's more than a few gigabytes in size, and you can't split the chunk because there is only one shard key value in the whole chunk. You can't just update all of the documents to use a more unique value because you can't update a shard key value. You could remove each document, change the shard key's value, and resave it, but that is not a swift operation for a large database.

The best you can do is start inserting "Asia, Japan" instead of just "Asia". Then you will have old documents that should be "Asia, Japan" but are just "Asia" and then your application logic will have to support both. Also, once you start having finer-grained chunks, there's no guarantee MongoDB will put them where you want (unless you turn off the balancer and do everything manually).

Data-center awareness is very important for large applications and it's a high priority for MongoDB developers. Choosing a low-cardinality shard key is not a good solution for the interim.

Ascending Shard Key

Reading data from RAM is faster than reading data from disk, so the goal is to have as much data as possible accessed in RAM. Thus, we want a shard key that keeps data together if it will be accessed together. For most applications, recent data is accessed more than older data, so people will often try to use something like a timestamp or `ObjectId` field for a shard key. This does not work as well as they expect it to.

Let's say we have a Twitter-like service where each document contains a short message, who sent it, and when it was sent. We shard on when it was sent, in seconds since the epoch.

We start out with one chunk: *(-∞, ∞)*, as always. All the inserts go to this shard until it splits into something like *(-∞, 1294516901), [1294516901, ∞)*. Now, we split chunks

at their midpoint, so when we split the chunk, the current timestamp is probably well after 1294516901. This means that all inserts will be going to the second chunk—nothing will be hitting the first anymore. Once the second chunk fills up, it'll be split up into *[1294516901, 1294930163)*, *[1294930163, ∞)*. But now the time will be after 1294930163, so everything will be added to the *[1294930163, ∞)* chunk. This pattern continues: everything will always be added to the "last" chunk, meaning everything will be added to one shard.

This shard key gives you a single, undistributable hot spot.

Keys that this rule applies to

This rule applies to anything ascending; it doesn't have to be a timestamp. This includes `ObjectId`s, dates, auto-incrementing primary keys (imported from other databases, possibly). If the key's values trend towards infinity, you will have this problem.

Exceptions to the rule

Basically, this shard key is always a bad idea because it guarantees you'll have a hotspot. If you have low traffic so that a single shard can handle almost all reads and writes, this could work. Of course, if you get a traffic spike or become more popular, it would stop working and be difficult to fix.

Don't use an ascending shard key unless you're sure you know what you're doing. There are better shard keys—this one should be avoided.

Random Shard Key

Sometimes, in an effort to avoid a hotspot, people choose a field with random values to shard by. This will work fine at first, but as you get more data, it can become slower and slower.

Let's say we're storing thumbnail photos in a sharded collection. Each document contains the binary data for the photo, an MD5 hash of the binary data, a description, the date it was taken, and who took it. We decide to shard on the MD5 hash.

As our collection grows, we end up with a pretty even number of chunks evenly distributed across our shards. So far so good. Now, let's say we're pretty busy and a chunk on Shard 2 fills up and splits. The configuration server notices that Shard 2 has 10 more chunks than Shard 1 and decides it should even things out. MongoDB now has to load a random five chunks' worth of data into memory and send it to Shard 1. This is data that wouldn't have been in memory ordinarily, because it's a completely random order of data. So, now MongoDB is going to be putting a lot more pressure on RAM and there's going to be a lot of disk IO going on (which is always slow).

You must have an index on the key you shard by, so if you choose a randomly-valued key that you don't query by, you're basically "wasting" an index. Every additional index makes writes slower, so it's important to keep the number of indexes as low as possible.

Good Shard Keys

What we really want is something that takes into account our access patterns. If our application is regularly accessing 25GB of data, we'd like to have all splits and migrates happen in the 25GB of data, not access data in a random pattern that must constantly copy new data from disk to memory.

So, we want to choose a shard key with nice data locality, but not so local that we end up with a hot spot.

Coarsely ascending key + search key

Many applications access new data more often than older data, so we want data to be roughly ordered by date, but also distributed evenly. This keeps the data we're reading and writing in memory, but distributes the load across the cluster.

We can accomplish this with a compound shard key—something like `{`*`coarselyAscending`*` : 1, `*`search`*` : 1}`. The *`coarselyAscending`* key should have between a few dozen and a few hundred chunks per value and the *`search`* key should be something that the application commonly queries by.

For example, let's say we have an analytics application where our users regularly access the last month worth of data; we want to keep that data handy. We'll shard on `{month : 1, user : 1}`. *month* is a coarsely ascending field, which means that every month it has a new, increasing value. *user* is a good second field because we'll often be querying for a certain user's data.

We'll start out with our one chunk, which now has a compound range: *((-∞,-∞), (∞, ∞))*. As it fills up, we split the chunk into two chunks, something like *((-∞, -∞), ("2011-04", "susan"))* and *[("2011-04", "susan"), (∞, ∞))*. Now, assuming it's still April (*"2011-04"*), the writes will be evenly distributed between the two chunks. All users with usernames less than "susan" will be put on the first chunk and all users with usernames greater than "susan" will be put on the second chunk.

As the data continues to grow, this continues to work. Subsequent chunks created in April will be moved to different shards, so reads and writes will be balanced across the cluster. In May, we start creating chunks with *"2011-05"* in their bounds. When June rolls around, we aren't accessing the data from *"04-2011"* chunks anymore, so these chunks can quietly drop out of memory and stop taking up resources. We might want to look at them again for historical reasons, but they should never need to be split or migrated (the problem we ran into with random indexes).

FAQ

Why not just use `{ascendingKey : 1}` *as the shard key?*
: This was covered in "Ascending Shard Key" and, if you combine it with a rough-grained ascending key, it can also create giant, unsplittable chunks.

Can `search` *be an ascending field, too?*
: No. If it is, the shard key will degenerate into an ascending key and you'll have the same hotspot problems a vanilla ascending key gives you.

So what should `search` *be?*
: Hopefully, the *search* field can be something useful your application can use for querying, possibly user info (as in the example above), a filename field, a GUID, etc. It should be a fairly randomly distributed non-ascending key with decent cardinality.

The general case

We can generalize this to a formula for shard keys:

```
{coarseLocality : 1, search : 1}
```

`coarseLocality` is whatever locality you want for your data. ***`search`*** is a common search on your data.

This key is not the only possible shard key and it won't work well for everything. However, it's a good way to start thinking about how to choose a shard key, even if you do not ultimately use it.

What shard key should I use?

Not knowing your application, I can't really tell you. Choosing a good shard key should take some work. Before you choose one, think about the answers to these questions:

- What do writes look like? What is the shape and size of the documents you're inserting?
- How much data is the system writing per hour? Per day? Peak?
- What fields are random, and which ones are increasing?
- What do reads look like? What data are people accessing?
- How much data is the system reading per hour? Per day? Peak?
- Is the data indexed? Should it be indexed?
- How much data is there, total?

There may be other patterns that you find in your data—use them! Before you shard, you should get to know your data very well.

Sharding a New or Existing Collection

Once you have selected a shard key, you are ready to shard your data.

Quick Start

If you're looking to set up a cluster to play around with as fast as possible, you can set one up in a minute or two using the mongo-snippets repository on Github. It's a little lacking in documentation, but this repository is basically a collection of useful scripts for users. Of particular interest is *simple-setup.py*, which automatically starts, configures, and populates a cluster (locally).

To run this script, you'll need to have the MongoDB Python driver. If you do not have it installed, you can install it (on *NIX systems) by running:

```
$ sudo easy_install pymongo
```

Once Pymongo is installed, download the *mongo-snippets* repository and run the following command:

```
$ python sharding/simple-setup.py --path=/path/to/your/mongodb/binaries
```

There are a number of other options available—run `python sharding/simple-setup.py --help` to see them. This script is a bit finicky, so make sure to use the full path (e.g., /home/user/mongo-install, not *~/mongo-install*).

simple-setup.py starts up a *mongos* on *localhost:27017*, so you can connect to it with the shell to play around.

If you are interested in setting up a serious-business cluster, read on.

Config Servers

You can run either one or three configuration servers. We'll be using three config servers, as that's what you should do whenever you're in production.

> "One or three" is a weird and somewhat arbitrary restriction, but the rationale is that one is good for testing, and three is good for production. Two is more than people would want to test with and not enough for production.
>
> You can't run an arbitrary number because their interactions are complex, and the programming and logic of what to do if *N* out of *M* config servers are down is difficult. MongoDB probably will support any number of config servers in the future, but it's not an immediate priority.

Setting up configuration servers is pretty boring because they're just vanilla *mongod* instances. The only important thing to note when setting up config servers is that you want some of them up all of the time, so try to put them in separate failure domains.

Let's say we have three data centers: one in New York, one in San Francisco, and one on the moon. We start up one config server in each center.

```
$ ssh ny-01
ny-01$ mongod

$ ssh sf-01
sf-01$ mongod

$ ssh moon-01
moon-01$ mongod
```

That's it!

You might notice a slight problem with this setup. These servers, which are supposed to be intimately connected to each other, have no idea that the others exist or that they are even config servers. This is fine—we're going to hook them up in a moment.

> Sometimes, people assume that they have to set up replication on config servers. Configuration servers do not use the same replication mechanism as "normal" *mongod* replication and should not be started as replica sets or master-slave setups. Just start config servers as normal, unconnected *mongod*s.

When you start up your config servers, feel free to use any of the options listed under "General Options" when you run *mongod --help*, except for `--keyFile` and `--auth`. Sharding does not support authentication at the moment, although this should change midway-through 1.9.

It's not a particularly good idea to run it with `--quiet` because you want to be able to figure out what's going on if something goes wrong. Instead of `--quiet`, use --logpath <file> and `--logappend` to send the logs somewhere so you'll have them if you run into problems.

The Sharding options apply to shards, not the configuration servers, so ignore those. You don't need to use any of the other options (under Replication, Replica set, or Master/slave) because you didn't start up your config servers with replication, right?

mongos

The next step is starting up a *mongos*. A sharded setup needs at least one *mongos* but can have as many as you'd like. Keep in mind that you will have to (or at least should) monitor all of the *mongos* processes you spin up, so you probably don't want thousands, but one *mongos* per application server usually is a good number. So, we'll log on to our application server and start up our first *mongos*.

```
$ ssh ny-02
ny-01$ mongos --configdb ny-01,sf-01,moon-01
```

Press Enter, and now all the config servers know about each other. The *mongos* is like the host of the party that introduces them all to each other.

You now have a trivial cluster. You can't actually store any data in it (configuration servers only store configuration, not data), but other than that, it's totally functional.

As you might want to use your database to read or write data, let's add some data storage.

Shards

All administration on a cluster is done through the *mongos*. So, connect your shell to the *mongos* process you started.

```
$ mongo ny-02:27017/admin
MongoDB shell version: 1.7.5
connecting to: admin
>
```

Make sure you're using the *admin* database. Setting up sharding requires commands to be run from the *admin* database.

Once you're connected, you can add a shard. There are two ways to add a shard, depending on whether the shard is a single server or a replica set. Let's say we have a single server, *sf-02*, that we've been using for data. We can make it the first shard by running the ***addShard*** command:

```
> db.runCommand({"addShard" : "sf-02:27017"})
{ "shardAdded" : "shard0000", "ok" : 1 }
```

This adds the server to the cluster. Now you can start storing and querying for data. (MongoDB will give you an error if you attempt to store data before you have any shards to put it on, for obvious reasons.)

In general, you should use a replica set, not a single server, for each shard. Replica sets will give you better availability. To add a replica set shard, you must give *addShard* a string of the form `"setName/seed1[,seed2[,seed3[,...]]]"`. That is, you must give the set name and at least one member of the set (*mongos* can derive the other members as long as it can connect to someone).

For example, if we had a replica set creatively named replica set "rs" with members *rs1-a*, *rs1-b*, and *rs1-c*, we could say:

```
> db.runCommand({"addShard" : "rs/rs1-a,rs1-c"})
{ "shardAdded" : "rs", "ok" : 1 }
```

Notice that we ended up with a much nicer name this time! (I was the one who programmed it to pick up on the replica set name, so I'm particularly proud of it.) If you add a replica set, its name will become the shard name.

You can name a shard anything you want. If you don't want the default, use the name option when you add a shard.

```
> db.runCommand({"addShard" : "sf-02:27017", "name" : "Golden Gate shard"})
{ "shardAdded" : "Golden Gate shard", "ok" : 1 }
> db.runCommand({"addShard" : "set1/rs1-a,rs1-b", "name" : "replicaSet1"})
{ "shardAdded" : "replicaSet1", "ok" : 1 }
```

You'll have to refer to shards by name occasionally (see Chapter 5), so don't make the name too crazy or long.

Limiting shard size

By default, MongoDB will evenly distribute data between shards. This is useful if you're using a bunch of commodity servers, but you can run into problems if you have one gargantuan machine with 10 terabytes and one ho-hum machine with a few hundred gigabytes. If your servers are seriously out of balance, you should use the `maxSize` option. This specifies the maximum size, in megabytes, that you want the shard to grow to.

Keep in mind that `maxSize` is more of a guideline than a rule. MongoDB will not cap off a shard at `maxSize` and not let it grow another byte, but rather it'll stop moving data to the shard and possibly move some data off. If it feels like it. So aim a bit low here.

maxSize is another option for the *addShard* command, so if you want to set a shard to only use 20GB, you could say:

```
> db.runCommand({"addShard" : "sfo/server1,server2", "maxSize" : 20000})
```

In the future, MongoDB will automatically figure out how much space it has to work with on each shard and plan accordingly. In the meantime, use `maxSize` to give it a hint.

Now you have a fully armed and operational cluster!

Databases and Collections

If you want MongoDB to distribute your data, you have to let it know which databases and collections you want to distribute. Start with the database. You have to tell MongoDB that a database can contain sharded collections before you try to shard its collections.

Let's say we're writing a blog application, so all of our collections live in the *blog* database. We can enable sharding on it with the command:

```
> db.adminCommand({"enableSharding" : "blog"})
{ "ok" : 1 }
```

Now we can shard a collection. To shard a collection, you must specify the collection and shard key. Suppose we were sharding on `{"date" : 1, "author" : 1}`.

```
> db.adminCommand({"shardCollection" : "blog.posts", key : {"date" : 1, "author" : 1}}
{ "collectionSharded" : "blog.posts", "ok" : 1 }
```

Note that we include the database name: *blog.posts*, not *posts*.

If we are sharding a collection with data in it, the data must have an index on the shard key. All documents must have a value for the shard key, too (and the value cannot be `null`). After you've sharded the collection, you can insert documents with a `null` shard key value.

Adding and Removing Capacity

As your application continues to grow, you'll need to add more shards. When you add a shard, MongoDB will start moving data from the existing shards to the new one. Keep in mind that moving data puts added pressure on shards. MongoDB tries to do this as gently as possible; it slowly moves one chunk at a time, then tries again later if the server seems busy. However, no matter how delicately MongoDB does it, moving chunks adds load.

This means that if you wait until your cluster is running at capacity to add more shards, adding a new shard can bring your application to a grinding halt through a chain reaction. Say your existing shards are just about maxed out, so you bring up a new shard to help with the load. The balancer pokes Shard 1 and asks it to send a chunk to the new shard. Shard 1 has just enough memory to keep your application's working set of data in RAM, and now it's going to have to pull a whole chunk into memory. The data that your application is using starts getting shunted out of RAM to make room for the chunk. This means that your application starts having to do disk seeks and MongoDB is doing disk seeks for chunk data. As queries start taking longer, requests begin to pile up, exacerbating the problem (see Figure 3-2).

The lesson? Add shards while you still have room to maneuver. If you no longer have that room, add shards in the middle of the night (or at another non-busy time). If you don't have a non-busy time and you waited too long, there are a couple of ways to hack around it, but they are neither fun nor easy. (You can manually create a Frankenstein shard from backups of your existing shard, manually change the *config.chunks* collection, and restart your whole cluster—this is obviously not recommended.) Add shards early and often.

So! How do you add shards? The same way that you added the first shard above, with the *addShard* command.

One interesting thing about adding subsequent shards is that they don't have to be empty. They cannot have databases that already exist in the cluster, but if your cluster has a *foo* database and you add a shard with a *bar* database, that's fine, as the config servers will pick up on it and it'll pop up in cluster info. Therefore, if you want, you can bring together a couple different systems into one large, sharded cluster.

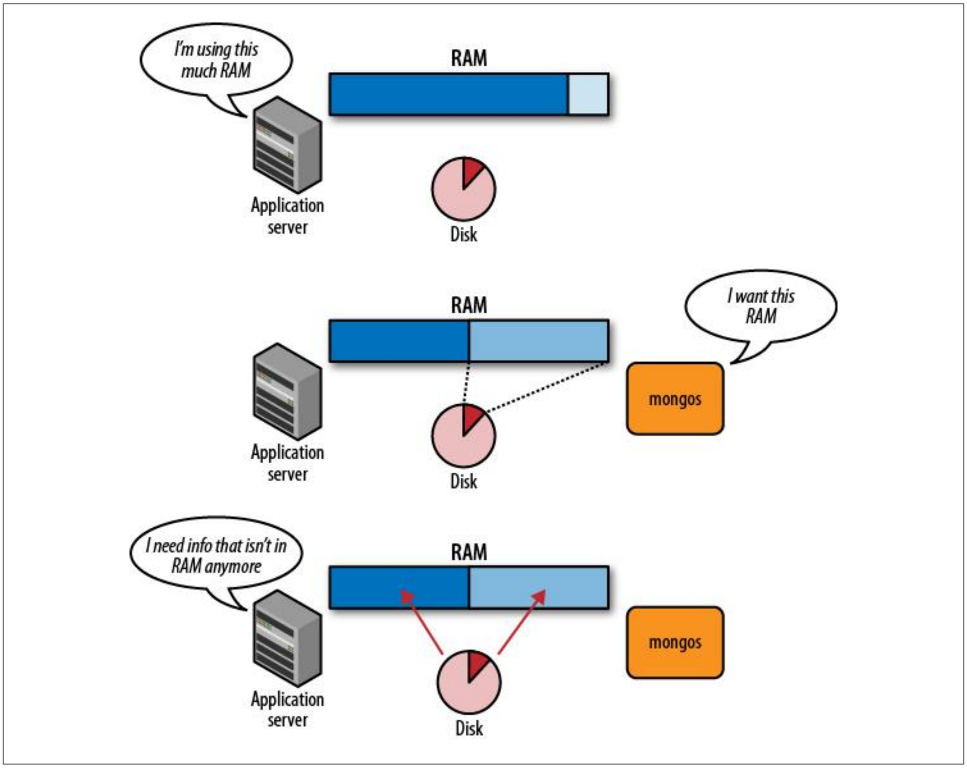


Figure 3-2. Moving chunks can force data out of RAM, making more requests hit disk and slowing everything down

Removing Shards

Sometimes removing shards comes up. Someone might shard too early or they realize that they chose the wrong shard key, so they want to get everything back onto one shard, dump, restore, and try again. You might just want to take certain servers offline.

The *removeShard* command lets you take a shard out of the cluster.

```
> db.runCommand({removeShard : "Golden Gate shard"})
{
    "msg" : "draining started successfully",
    "state" : "started",
    "shard" : "Golden Gate shard",
    "ok" : 1
}
```

Note that the message says “started successfully.” It has to move all of the information that was on this shard to other shards before it can remove this shard. As mentioned above, moving data from shard to shard is pretty slow. The *removeShard* command returns immediately, and you must poll it to find out if it has finished. If you call it again, you'll see something like:

```
> db.runCommand({removeShard : "Golden Gate shard"})
{
    "msg" : "draining ongoing",
    "state" : "ongoing",
    "remaining" : {
        "chunks" : NumberLong(2),
        "dbs" : NumberLong(1)
    },
    "ok" : 1
}
```

Once it's done, its status will change to "completed." Then, you can safely shut down the shard (or use it as a non-sharded MongoDB server).

```
> db.runCommand({removeShard : "Golden Gate shard"})
{
    "msg" : "removeshard completed successfully",
    "state" : "completed",
    "shard" : "Golden Gate shard",
    "ok" : 1
}
```

This means that shard has been completely drained and may be shut down or used for other purposes.

Changing Servers in a Shard

If you're using a replica set, you can add or remove servers from the replica set and your *mongos* will pick up on the change. To modify your replica set in a cluster, do the exact same thing you would do if it was running independently: connect to the primary (*not* through *mongos*) and make any configuration changes you need.

CHAPTER 4
Working With a Cluster

Querying a MongoDB cluster is usually identical to querying a single *mongod*. However, there are some exceptions that are worth knowing about.

Querying

If you are using replica sets as shards and a *mongos* version 1.7.4 or more recent, you can distribute reads to slaves in a cluster. This can be handy for handling read load, although the usual caveats on querying slaves apply: you must be willing to get older data.

To query a slave through *mongos*, you must set the "slave okay" option (basically checking off that you're okay with getting possibly out-of-date data) with whatever driver you're using. In the shell, this looks like:

```
> db.getMongo().setSlaveOk()
```

Then query the *mongos* normally.

"Why Am I Getting This?"

When you work with a cluster, you lose the ability to examine an entire collection as a single "snapshot in time." Many people don't realize the ramifications of this until it hits them in the nose, so we'll go over some of the common ways it can affect applications.

Counting

When you do a *count* on a sharded collection, you may not get the results you expect. You may get quite a few more documents than actually exist.

The way a *count* works is the *mongos* forwards the `count` command to every shard in the cluster. Then, each shard does a `count` and sends its results back to the *mongos*,

which totals them up and sends them to the user. If there is a migration occurring, many documents can be present (and thus counted) on more than one shard.

When MongoDB migrates a chunk, it starts copying it from one shard to another. It still routes all reads and writes to that chunk to the old shard, but it is gradually being populated on the other shard. Once the chunk has finished "moving," it actually exists on both shards. As the final step, MongoDB updates the config servers and deletes the copy of the data from the original shard (see Figure 4-1).

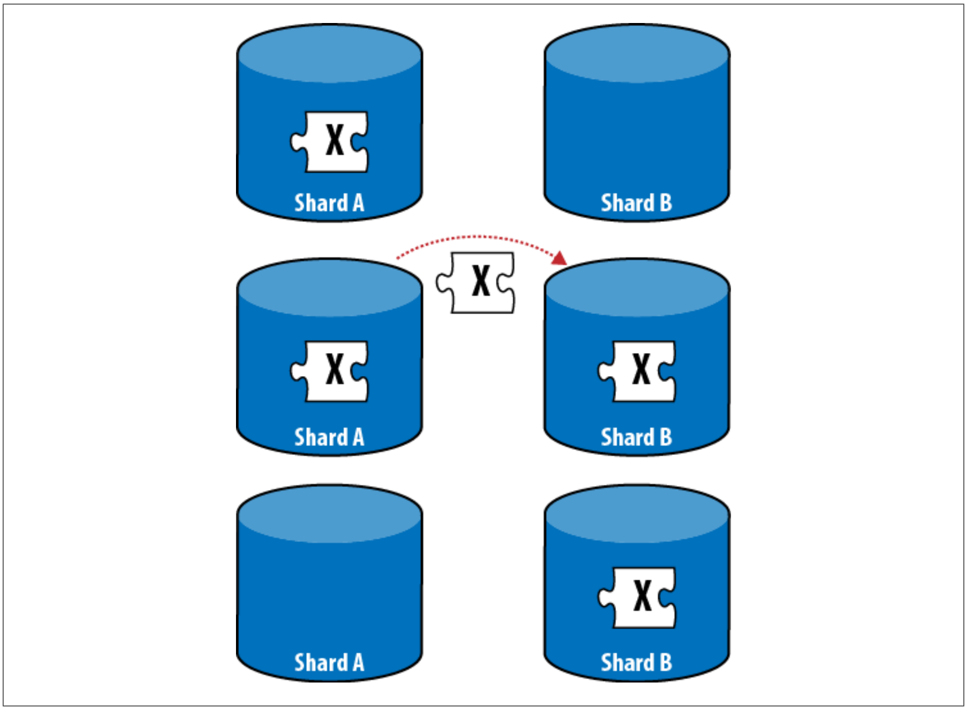


Figure 4-1. A chunk is migrated by copying it to the new shard, then deleting it from the shard it came from

Thus, when data is counted, it ends up getting counted twice. MongoDB may hack around this in the future, but for now, keep in mind that counts may overshoot the actual number of documents.

Unique Indexes

Suppose we were sharding on *email* and wanted to have a unique index on *username*. This is not possible to enforce with a cluster.

Let's say we have two application servers processing users. One application server adds a new user document with the following fields:

```
{
    "_id" : ObjectId("4d2a2e9f74de15b8306fe7d0"),
    "username" : "andrew",
    "email" : "awesome.guy@example.com"
}
```

The only way to check that "andrew" is the only "andrew" in the cluster is to go through every username entry on every machine. Let's say MongoDB goes through all the shards and no one else has an "andrew" username, so it's just about to write the document on Shard 3 when the second appserver sends this document to be inserted:

```
{
    "_id" : ObjectId("4d2a2f7c56d1bb09196fe7d0"),
    "username" : "andrew",
    "email" : "cool.guy@example.com"
}
```

Once again, every shard checks that it has no users with username "andrew". They still don't because the first document hasn't been written yet, so Shard 1 goes ahead and writes this document. Then Shard 3 finally gets around to writing the first document. Now there are two people with the same username!

The only way to guarantee no duplicates between shards in the general case is to lock down the entire cluster every time you do a write until the write has been confirmed successful. This is not performant for a system with a decent rate of writes.

Therefore, you cannot guarantee uniqueness on any key other than the shard key. You can guarantee uniqueness on the shard key because a given document can only go to one chunk, so it only has to be unique on that one shard, and it'll be guaranteed unique in the whole cluster. You can also have a unique index that is prefixed by the shard key. For example, if we sharded the users collection on *username*, as above, but with the unique option, we could create a unique index on `{username : 1, email : 1}`.

One interesting consequence of this is that, unless you're sharding on *_id*, you can create non-unique *_id*s. This isn't recommended (and it can get you into trouble if chunks move), but it is possible.

Updating

Updates, by default, only update a single record. This means that they run into the same problem unique indexes do: there's no good way of guaranteeing that something happens once across multiple shards. If you're doing a single-document update, it must use the shard key in the criteria (`update`'s first argument). If you do not, you'll get an error.

```
> db.adminCommand({shardCollection : "test.x", key : {"y" : 1}})
{ "shardedCollection" : "test.x", "ok" : 1 }
>
> // works okay
> db.x.update({y : 1}, {$set : {z : 2}}, true)
>
```

```
> // error
> db.x.update({z : 2}, {$set : {w : 4}})
can't do non-multi update with query that doesn't have the shard key
```

You can do a multiupdate using any criteria you want.

```
> db.x.update({z : 2}, {$set : {w : 4}}, false, true)
> // no error
```

If you run across an odd error message, consider whether the operation you're trying to perform would have to atomically look at the entire cluster. Such operations are not allowed.

MapReduce

When you run a MapReduce on a cluster, each shard performs its own map and reduce. *mongos* chooses a "leader" shard and sends all the reduced data from the other shards to that one for a final reduce. Once the data is reduced to its final form, it will be output in whatever method you've specified.

As sharding splits the job across multiple machines, it can perform MapReduces faster than a single server. However, it still isn't meant for real-time calculations.

Temporary Collections

In 1.6, MapReduce created temporary collections unless you specified the "out" option. These temporary collections were dropped when the connection that created them was closed. This worked well on a single server, but *mongos* keeps its own connection pools and never closes connections to shards. Thus, temporary collections were never cleaned up (because the connection that created them never closed), and they would just hang around forever, growing more and more numerous.

If you're running 1.6 and doing MapReduces, you'll have to manually clean up your temporary collections. You can run the following function to delete all of the temporary collections in a given database:

```
var dropTempCollections = function(dbName) {
    var target = db.getSisterDB(dbName);
    var names = target.getCollectionNames();

    for (var i = 0; i < names.length; i++) {
        if (names[i].match(/tmp\.mr\./)){
            target[names[i]].drop();
        }
    }
}
```

In later versions, MapReduce forces you to choose to do something with your output. See the documentation for details.

CHAPTER 5
Administration

Whereas the last chapter covered working with MongoDB from an application developer's standpoint, this chapter covers some more operational aspects of running a cluster. Once you have a cluster up and running, how do you know what's going on?

Using the Shell

As with a single instance of MongoDB, most administration on a cluster can be done through the *mongo* shell.

Getting a Summary

`db.printShardingStatus()` is your executive summary. It gathers all the important information about your cluster and presents it nicely for you.

```
> db.printShardingStatus()
--- Sharding Status ---
sharding version: { "_id" : 1, "version" : 3 }
shards:
{ "_id" : "shard0000", "host" : "ubuntu:27017" }
{ "_id" : "shard0001", "host" : "ubuntu:27018" }
databases:
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "test", "partitioned" : true, "primary" : "shard0000" }
test.foo chunks:
shard0001 15
shard0000 16
{ "_id" : { $minKey : 1 } } -->> { "_id" : 0 } on : shard1 { "t" : 2, "i" : 0 }
{ "_id" : 0 } -->> { "_id" : 15074 } on : shard1 { "t" : 3, "i" : 0 }
{ "_id" : 15074 } -->> { "_id" : 30282 } on : shard1 { "t" : 4, "i" : 0 }
{ "_id" : 30282 } -->> { "_id" : 44946 } on : shard1 { "t" : 5, "i" : 0 }
{ "_id" : 44946 } -->> { "_id" : 59467 } on : shard1 { "t" : 7, "i" : 0 }
{ "_id" : 59467 } -->> { "_id" : 73838 } on : shard1 { "t" : 8, "i" : 0 }
... some lines omitted ...
{ "_id" : 412949 } -->> { "_id" : 426349 } on : shard1 { "t" : 6, "i" : 4 }
{ "_id" : 426349 } -->> { "_id" : 457636 } on : shard1 { "t" : 7, "i" : 2 }
```

```
37
{ "_id" : 457636 } -->> { "_id" : 471683 } on : shard1 { "t" : 7, "i" : 4 }
{ "_id" : 471683 } -->> { "_id" : 486547 } on : shard1 { "t" : 7, "i" : 6 }
{ "_id" : 486547 } -->> { "_id" : { $maxKey : 1 } } on : shard1 { "t" : 7, "i" : 7 }
```

`db.printShardingStatus()` prints a list of all of your shards and databases. Each sharded collection has an entry (there's only one sharded collection here, *test.foo*). It shows you how chunks are distributed (15 chunks on `shard0001` and 16 chunks on `shard0000`). Then it gives detailed information about each chunk: its range—e.g., `{ "_id" : 115882 } -->> { "_id" : 130403 }` corresponding to *_id*s in *[115882, 130403)*—and what shard it's on. It also gives the major and minor version of the chunk, which you don't have to worry about.

Each database created has a primary shard that is its "home base." In this case, the *test* database was randomly assigned shard0000 as its home. This doesn't really mean anything—shard0001 ended up with more chunks than shard0000! This field should never matter to you, so you can ignore it. If you remove a shard and some database has its "home" there, that database's home will automatically be moved to a shard that's still in the cluster.

`db.printShardingStatus()` can get really long when you have a big collection, as it lists every chunk on every shard. If you have a large cluster, you can dive in and get more precise information, but this is a good, simple overview when you're starting out.

The config Collections

mongos forward your requests to the appropriate shard—except for when you query the *config* database. Accessing the *config* database patches you through to the config servers, and it is where you can find all the cluster's configuration information. If you do have a collection with hundreds or thousands of chunks, it's worth it to learn about the contents of the *config* database so you can query for specific info, instead of getting a summary of your entire setup.

Let's take a look at the *config* database. Assuming you have a cluster set up, you should see these collections:

```
> use config
switched to db config
> show collections
changelog
chunks
collections
databases
lockpings
locks
mongos
settings
shards
system.indexes
version
```

Many of the collections are just accounting for what's in the cluster:

config.mongos
: A list of all *mongos* processes, past and present

```
> db.mongos.find()
{ "_id" : "ubuntu:10000", "ping" : ISODate("2011-01-08T10:11:23"), "up" : 0 }
{ "_id" : "ubuntu:10000", "ping" : ISODate("2011-01-08T10:11:23"), "up" : 20 }
{ "_id" : "ubuntu:10000", "ping" : ISODate("2011-01-08T10:11:23"), "up" : 1 }
```

_id is the hostname of the *mongos*. *ping* is the last time the config server pinged it. *up* is whether it thinks the *mongos* is up or not. If you bring up a *mongos*, even if it's just for a few seconds, it will be added to this list and will never disappear. It doesn't really matter, it's not like you're going to be bringing up millions of *mongos* servers, but it's something to be aware of so you don't get confused if you look at the list.

config.shards
: All the shards in the cluster

config.databases
: All the databases, sharded and non-sharded

config.collections
: All the sharded collections

config.chunks
: All the chunks in the cluster

config.settings contains (theoretically) tweakable settings that depend on the database version. Currently, *config.settings* allows you to change the chunk size (but don't!) and turn off the balancer, which you usually shouldn't need to do. You can change these settings by running an update. For example, to turn off the balancer:

```
> db.settings.update({"_id" : "balancer"}, {"$set" : {"stopped" : true }}, true)
```

If it's in the middle of a balancing round, it won't turn off until the current balancing has finished.

The only other collection that might be of interest is the *config.changelog* collection. It is a *very* detailed log of every split and migrate that happens. You can use it to retrace the steps that got your cluster to whatever its current configuration is. Usually it is more detail than you need, though.

"I Want to Do X, Who Do I Connect To?"

If you want to do any sort of normal reads, writes, or administration, the answer is always "a *mongos*." It can be any *mongos* (remember that they're stateless), but it's always a *mongos*—not a shard, not a config server.

You might connect to a config server or a shard if you're trying to do something unusual. This might be looking at a shard's data directly or manually editing a messed up

configuration. For example, you'll have to connect directly to a shard to change a replica set configuration.

Remember that config servers and shards are just normal *mongod*s; anything you know how to do on a *mongod* you can do on a config server or shard. However, in the normal course of operation, you should almost never have to connect to them. All normal operations should go through *mongos*.

Monitoring

Monitoring is crucially important when you have a cluster. All of the advice for monitoring a single node applies when monitoring many nodes, so make sure you have read the documentation on monitoring.

Don't forget that your network becomes more of a factor when you have multiple machines. If a server says that it can't reach another server, investigate the possibility that the network between two has gone down.

If possible, leave a shell connected to your cluster. Making a connection requires MongoDB to briefly give the connection a lock, which can be a problem for debugging. Say a server is acting funny, so you fire up a shell to look at it. Unfortunately, the *mongod* is stuck in a write lock, so the shell will sit there forever trying to acquire the lock and never finish connecting. To be on the safe side, leave a shell open.

mongostat

mongostat is the most comprehensive monitoring available. It gives you tons of information about what's going on with a server, from load to page faulting to number of connections open.

If you're running a cluster, you can start up a separate *mongostat* for every server, but you can also run *mongostat --discover* on a *mongos* and it will figure out every member of the cluster and display their stats.

For example, if we start up a cluster using the *simple-setup.py* script described in Chapter 4, it will find all the *mongos* processes and all of the shards:

```
$ mongostat --discover
                mapped  vsize    res faults locked % idx miss %  conn       time repl
localhost:27017     0m   105m     3m      0        0          0     2   22:59:50  RTR
localhost:30001    80m   175m     5m      0        0          0     3   22:59:50
localhost:30002     0m    95m     5m      0        0          0     3   22:59:50
localhost:30003     0m    95m     5m      0        0          0     3   22:59:50

localhost:27017     0m   105m     3m      0        0          0     2   22:59:51  RTR
localhost:30001    80m   175m     5m      0        0          0     3   22:59:51
localhost:30002     0m    95m     5m      0        0          0     3   22:59:51
localhost:30003     0m    95m     5m      0        0          0     3   22:59:51
```

I've simplified the output and removed a number of columns because I'm limited to 80 characters per line and *mongostat* goes a good 166 characters wide. Also, the spacing is a little funky because the tool starts with "normal" *mongostat* spacing, figures out what the rest of the cluster is, and adds a couple more fields: *qr|qw* and *ar|aw*. These fields show how many connections are queued for reads and writes and how many are actively reading and writing.

The Web Admin Interface

If you're using replica sets for shards, make sure you start them with the `--rest` option. The web admin interface for replica sets (*http://localhost:28017/_replSet*, if *mongod* is running on port 27017) gives you loads of information.

Backups

Taking backups on a running cluster turns out to be a difficult problem. Data is constantly being added and removed by the application, as usual, but it's also being moved around by the balancer. If you take a dump of a shard today and restore it tomorrow, you may have the same documents in two places or end up missing some documents altogether (see Figure 5-1).

The problem with taking backups is that you usually only want to restore parts of your cluster (you don't want to restore the entire cluster from yesterday's backup, just the node that went down). If you restore data from a backup, you have to be careful. Look at the config servers and see which chunks are supposed to be on the shard you're restoring. Then only restore data from those chunks using your backups (and *mongorestore*).

If you want a snapshot of the whole cluster, you would have to turn off the balancer, *fsync* and lock the slaves in the cluster, take dumps from them, then unlock them and restart the balancer. Typically people just take backups from individual shards.

Suggestions on Architecture

You can create a sharded cluster and leave it at that, but what happens when you want to do routine maintenance? There are a few extra pieces you can add that will make your setup easier to manage.

Create an Emergency Site

The name implies that you're running a website, but this applies to most types of application. If you need to bring your application down occasionally (e.g., to do maintenance, roll out changes, or in an emergency), it's very handy to have an emergency site that you can switch over to.

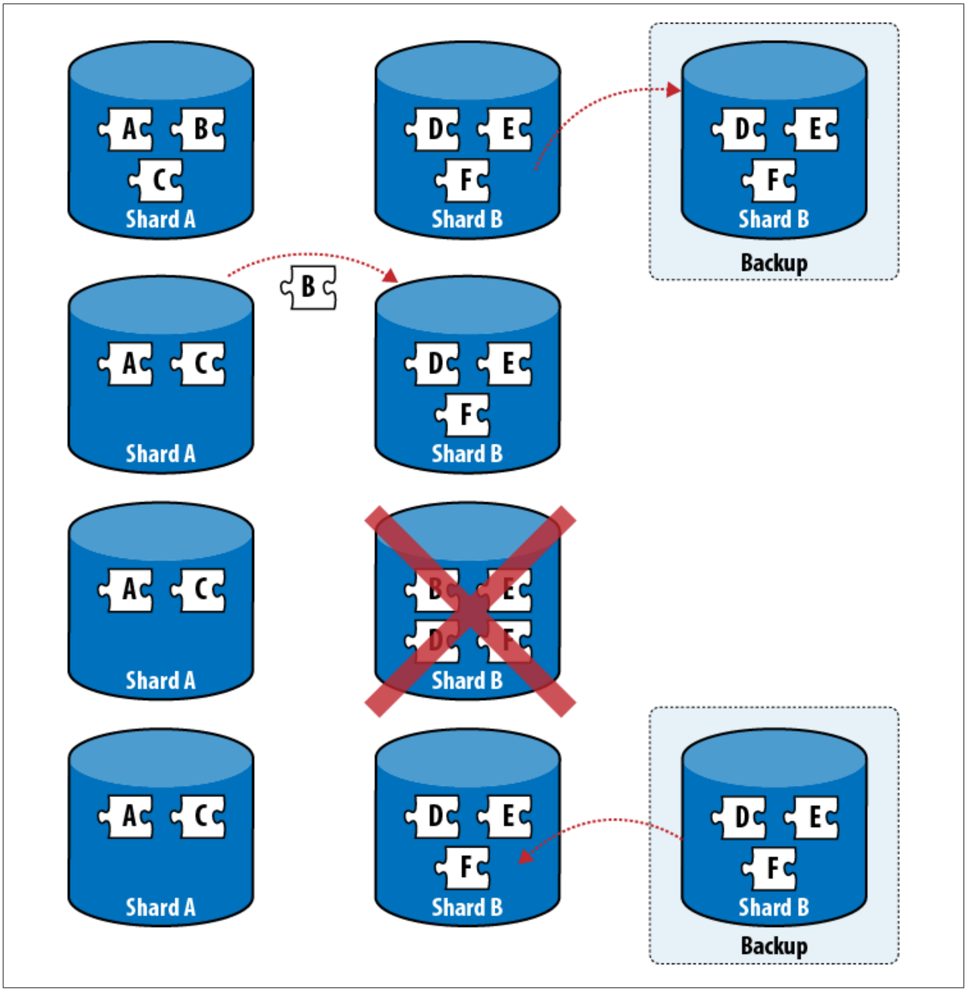


Figure 5-1. Here, a backup is taken before a migrate. If the shard crashes after the migrate is complete and restored from backup, the cluster will be missing the migrated chunk.

The emergency site should not use your cluster at all. If it uses a database, it should be completely disconnected from your main database. You could also have it serve data from a cache or be a completely static site, depending on your application. It's a good idea to set up something for users to look at, though, other than an Apache error page.

Create a Moat

A excellent way to prevent or minimize all sorts of problems is to create a virtual moat around your machines and control access to the cluster via a queue.

A queue can allow your application to continue handling writes in a planned outage, or at least prevent any writes that didn't quite make it before the outage from getting lost. You can keep them on the queue until MongoDB is up again and then send them to the *mongos*.

A queue isn't only useful for disasters—it can also be helpful in regulating bursty traffic. A queue can hold the burst and release a nice, constant stream of requests, instead of allowing a sudden flood to swamp the cluster. You can also use a queue going the other way: to cache results coming out of MongoDB.

There are lots of different queues you could use: Amazon's SQS, RabbitMQ, or even a MongoDB capped collection (although make sure it's on a separate server than the cluster it's protecting). Use whatever queue you're comfortable with.

Queues won't work for all applications. For example, they don't work with applications that need real-time data. However, if you have an application that can stand small delays, a queue can be useful intermediary between the world and your database.

What to Do When Things Go Wrong

As mentioned in the first chapter, network partitions, server crashes, and other problems can cause a whole variety of issues. MongoDB can "self-heal," at least temporarily, from many of these issues. This section covers which outages you can sleep through and which ones you can't, as well as preparing your application to deal with outages.

A Shard Goes Down

If an entire shard goes down, reads and writes that would have hit that shard will return errors. Your application should handle those errors (it'll be whatever your language's equivalent of an exception is, thrown as you iterated through a cursor). For example, if the first three results for some query were on the shard that is up and the next shard containing useful chunks is down, you'd get something like:

```
> db.foo.find()
{ "_id" : 1 }
{ "_id" : 2 }
{ "_id" : 3 }
error: mongos connectionpool:
connect failed ny-01:10000 : couldn't connect to server ny-01:10000
```

Be prepared to handle this error and keep going gracefully. Depending on your application, you could also do exclusively targeted queries until the shard comes back online.

Support will be added for partial query results in the future (post-1.8.0), which will only return results from shards that are up and not indicate that there were any problems.

Most of a Shard Is Down

If you are using replica sets for shards, hopefully an entire shard won't go down, but merely a server or two in the set. If the set loses a majority of its members, no one will be able to become master (without manual rejiggering), and so the set will be read-only. If a set becomes read-only, make sure your application is only sending it reads and using `slaveOkay`.

If you're using replica sets, hopefully a single server (or even a few servers) failing won't affect your application at all. The other servers in the set will pick up the slack and your application won't even notice the change.

> In 1.6, if a replica set configuration changes, there may be a zillion identical messages printed to the log. Every connection between *mongos* and the shard prints a message when it notices that its replica set connection is out-of-date and updates it. However, it shouldn't have an impact on what's actually happening—it's just a lot of sound and fury. This has been fixed for 1.8; *mongos* is much smarter about updating replica set configurations.

Config Servers Going Down

If a config server goes down, there will be no immediate impact on cluster performance, but no configuration changes can be made. All the config servers work in concert, so none of the other config servers can make any changes while even a single of their brethren have fallen. The thing to note about config servers is that no configuration can change while a config server is down—you can't add mongos servers, you can't migrate data, you can't add or remove databases or collections, and you can't change replica set configurations.

If a config server crashes, do get it back up so that your config can change when it needs to, but it shouldn't affect the immediate operation of your cluster at all. Make sure you monitor config servers and, if one fails, get it right back up.

Having a config server go down can put some pressure on your servers if there is a migrate in progress. One of the last steps of the migrate is to update the config servers. Because one server is down, they can't be updated, so the shards will have to back out the migration and delete all the data they just painstakingly copied. If your shards aren't overloaded, this shouldn't be too painful, but it is a bit of a waste.

Mongos Processes Going Down

As you can always have extra *mongos* processes and they have no state, it's not too big a deal if one goes down. The recommended setup is to run one *mongos* on each appserver and have each appserver talk to its local *mongos* (Figure 5-2). Then, if the whole machine goes down, no one is trying to talk to a *mongos* that isn't there.

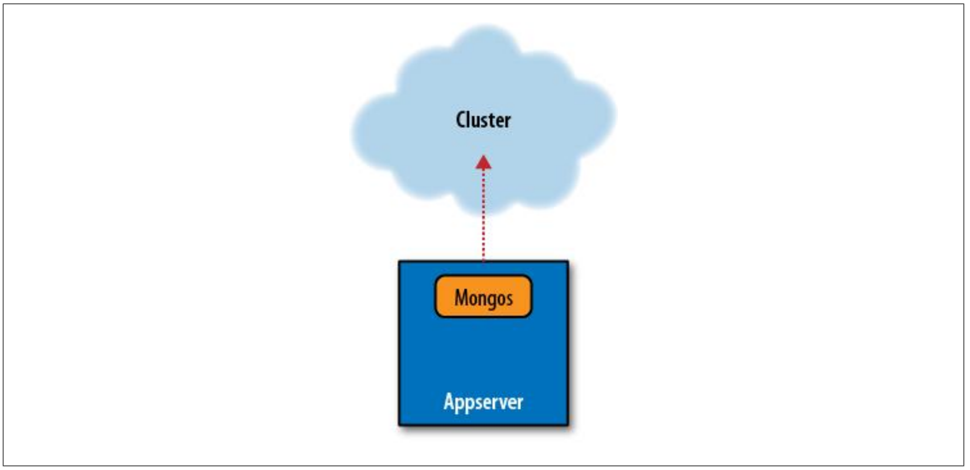


Figure 5-2. An appserver running a mongos.

Have a couple extra *mongos* servers out there that you can fail over to if one *mongos* process crashes while the application server is still okay. Most drivers let you specify a list of servers to connect to and will try them in order. So, you could specify your preferred *mongos* first, then your backup *mongos*. If one goes down, your application can handle the exception (in whatever language you're using) and the driver will automatically shunt the application over to your backup *mongos* for the next request.

You can also just try restarting a crashed *mongos* if the machine is okay, as they are stateless and store no data.

Other Considerations

Each of the points above is handled in isolation from anything else that could go wrong. Sometimes, if you have a network partition, you might lose entire shards, parts of other shards, config servers, and mongos processes. You should think carefully about how to handle various scenarios from both user-facing (will users still be able to do anything?) and application-design (will the application still do something sensible?) perspectives.

Finally, MongoDB tries to let a lot go wrong before exposing a loss of functionality. If you have the perfect storm (and you will), you'll lose functionality, but day-to-day server crashes, power outages, and network partitions shouldn't cause huge problems. Keep an eye on your monitoring and don't panic.

CHAPTER 6
Further Reading

If you follow the advice in the preceding chapters, you should be well on your way to an efficient and predictable distributed system that can grow as you need. If you have further questions or are confused about anything, feel free to email me at *kristina@10gen.com*.

If you're interested in learning more about sharding, there are quite a few resources available:

- The MongoDB wiki has a large section on sharding, with everything from configuration examples to discussions of internals.
- The MongoDB user list is a great place to ask questions.
- There are lots of useful little pieces of code in the mongo-snippets repository.
- Boxed Ice runs a production MongoDB cluster and often writes useful articles in their blog about running MongoDB.
- If you're interested in reading more about distributed computing theory, I highly recommend Leslie Lamport's original Paxos paper, which is an entertaining and instructive read.

Also, if you enjoyed this, I write a blog that mostly covers advanced MongoDB topics.